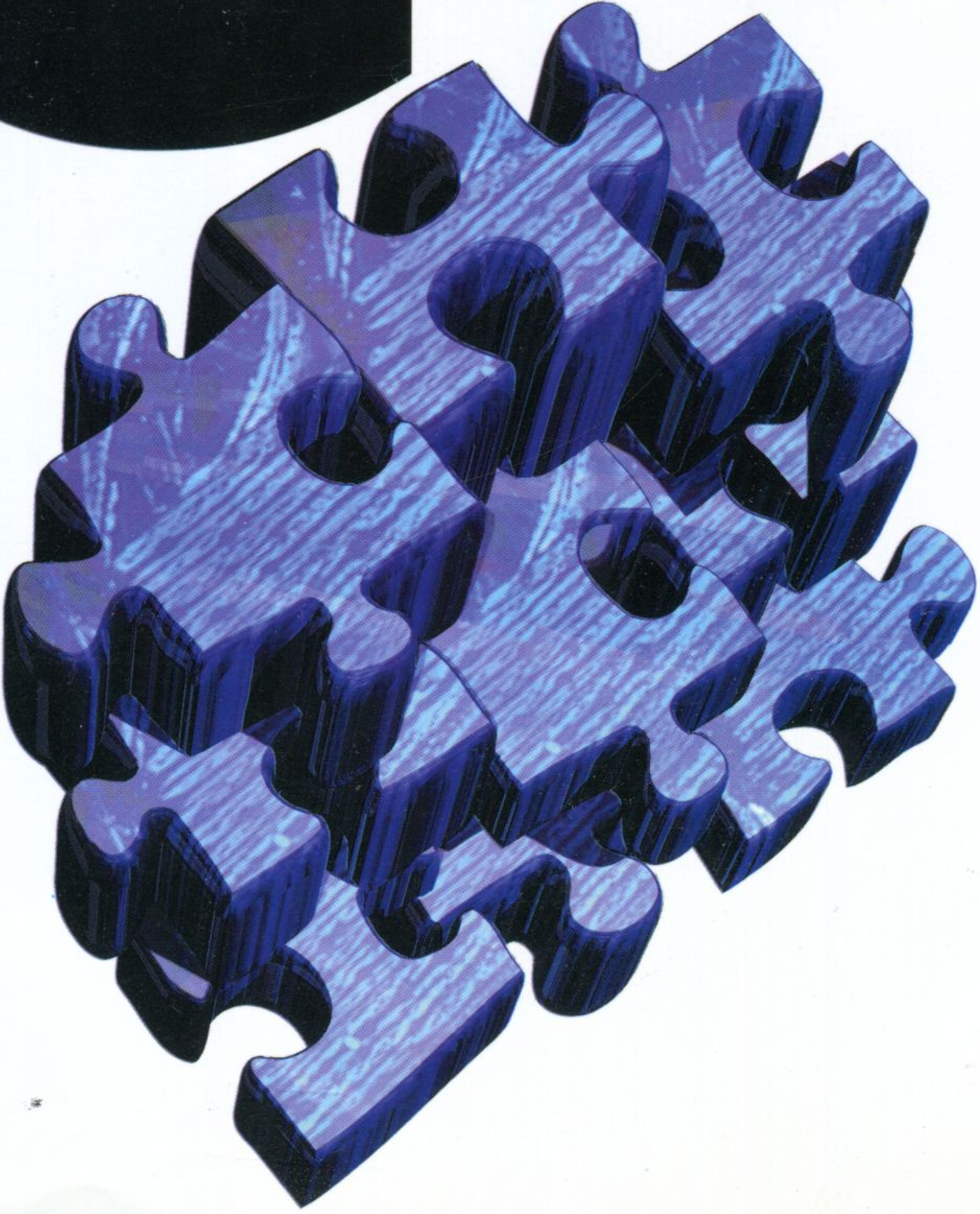
Software Technology 25

決定版
はじめてのC++

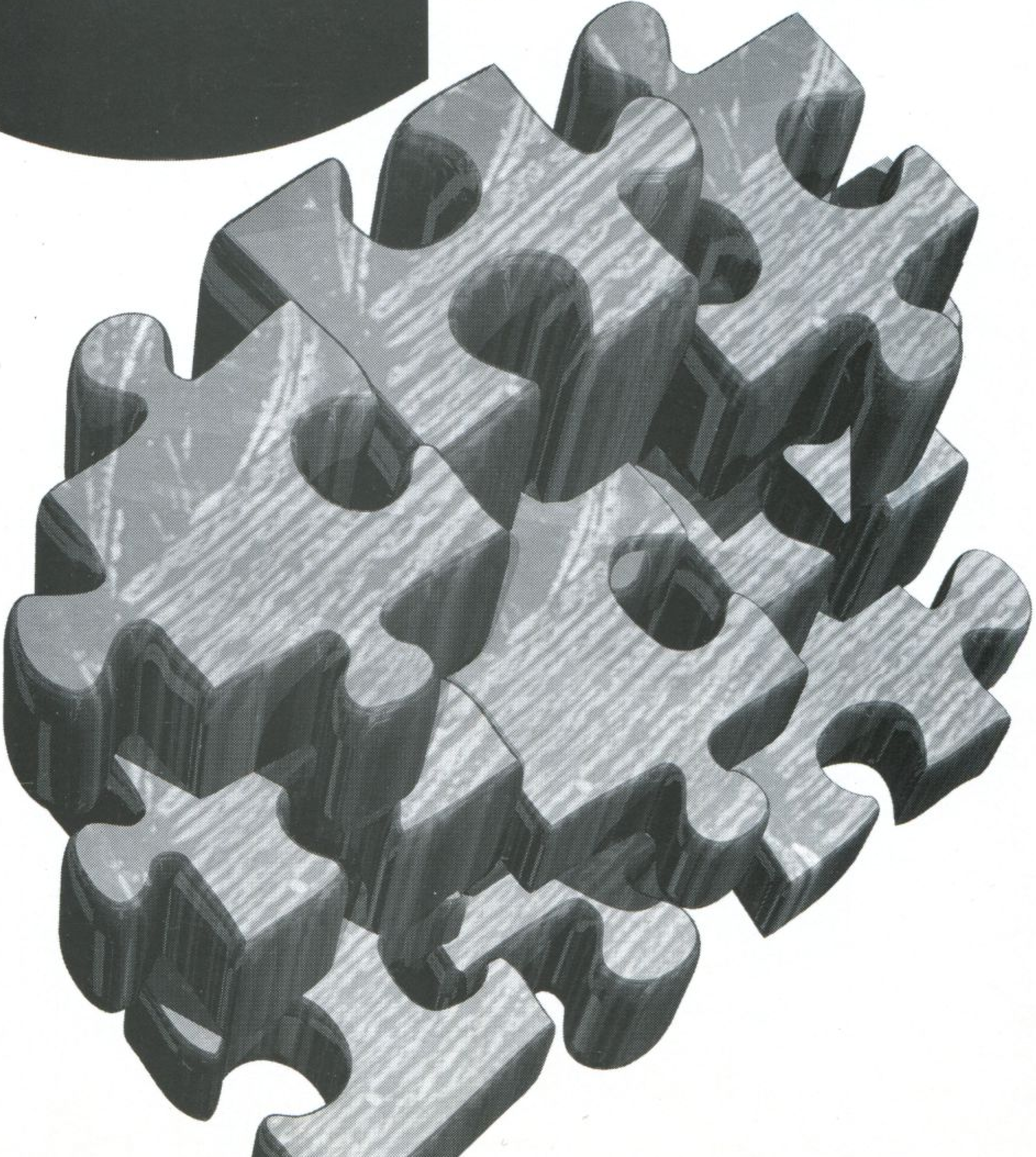
塚越一雄／著　技術評論社



Software Technology 25

決定版 はじめての C++

塚越一雄／著　技術評論社



Software Technology 25

決定版 はじめてのC++

塚越一雄／著

技術評論社

本書の執筆環境　　　：Mule＋Canna
本書で使用した処理系：GNU g++／gcc version 2.7.2.3＋ncurses

はじめに

C++は難解な言語だといわれる．これは，C++がCを基礎に作られていることと関係があるからかもしれない．多くのC++の本は，あらかじめCを知っているものとして書かれていることが多い．このため，読者はまずCの習得に時間を取られ，あまりの遠い道のりに挫折してしまう．

幸い，本書はCの知識を前提にしていない．したがって，プログラムを書いたことがなくても，C++のプログラミングを学ぶことができる．

できれば，一度，CもしくはC++に挫折した上で本書を手にしていただきたい．そして，第1章の始めから順に読んでいただきたい．すらすら読めるので驚くだろう．

この本は，全部で3部構成になっている．次に各部の構成を示す．

《第1部　基礎》

プログラミングの入口からオブジェクト指向プログラミングの手前までのC++の基礎を扱う．いわゆる「C＋ベターC」に相当する部分である．オブジェクト指向を必要としないプログラミングなら，第1部の内容だけで十分である．

《第2部　オブジェクト指向プログラミング》

タイトルどおり，C++の特色であるオブジェクト指向プログラミングを取り上げる．オブジェクト指向プログラミングの真意は**多態性**にある．したがって第23章以降が，この部の中心になる．

《第3部　応用プログラミング》

第2部までの基礎を踏まえて，C++の応用プログラミングを取り上げる．最も面白いところでもある．特に，最終章の「**イベント駆動型アプリケーションフレームワーク**」は，C++の特色が最もよく活かされたところなので，堪能されたい．

テーマの関係でUNIXに依存している章がある．もしUNIXの環境を持っていないなら，PC上にフリーのUNIXをインストールしてテストされることをお勧めする．

第3部の各章は独立しているので，興味のある章から読破されたい．

末筆ながら，本書の執筆にあたりお世話になった技術評論社第2編集部の跡部和之氏にお礼申し上げます．

1999年8月

塚越一雄

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

基　礎

PART 1

第1章

旅立ち——はじめてのコンパイル

この章の目的は，**コンパイル可能なC++のプログラム**を作成することである．これ以上削ったらコンパイルできないという最も小さなC++のプログラムを作り，それをコンパイルする．

1 プログラムの構成

これからC++のプログラミングをマスターするための旅に出発する．

手始めに，最も短いC++のプログラムを書く．これ以上削ったらC++のプログラムにならないというギリギリのプログラムから始める．

1.1◆C++のプログラム

C++は，コンピュータプログラムを書くための言語である．一般に，プログラムは，その言語が理解できる文を並べることによって構成される．

```
文1
文2
文3
……
```

ただし，文を配置する規則は言語によって異なる．BASICなどの原始的な言語は，文をそのまま並べれば，それがプログラムになる．つまり，次は立派なBASICのプログラムである．

BASICの文1
BASICの文2
BASICの文3
……………

また，極端な話，文が1つだけというBASICのプログラムも存在する．

BASICの文◀――これだけでプログラムが完結している

一方，C++の文は，裸のままでは並べることができない．文は，**関数**というブロック（文のかたまり）の中に閉じ込められる．

関数
　文1
　文2
　文3
　……

したがって，C++のプログラムの最小単位は関数である．「C++のプログラムは，関数を並べたものである」ということもできる．

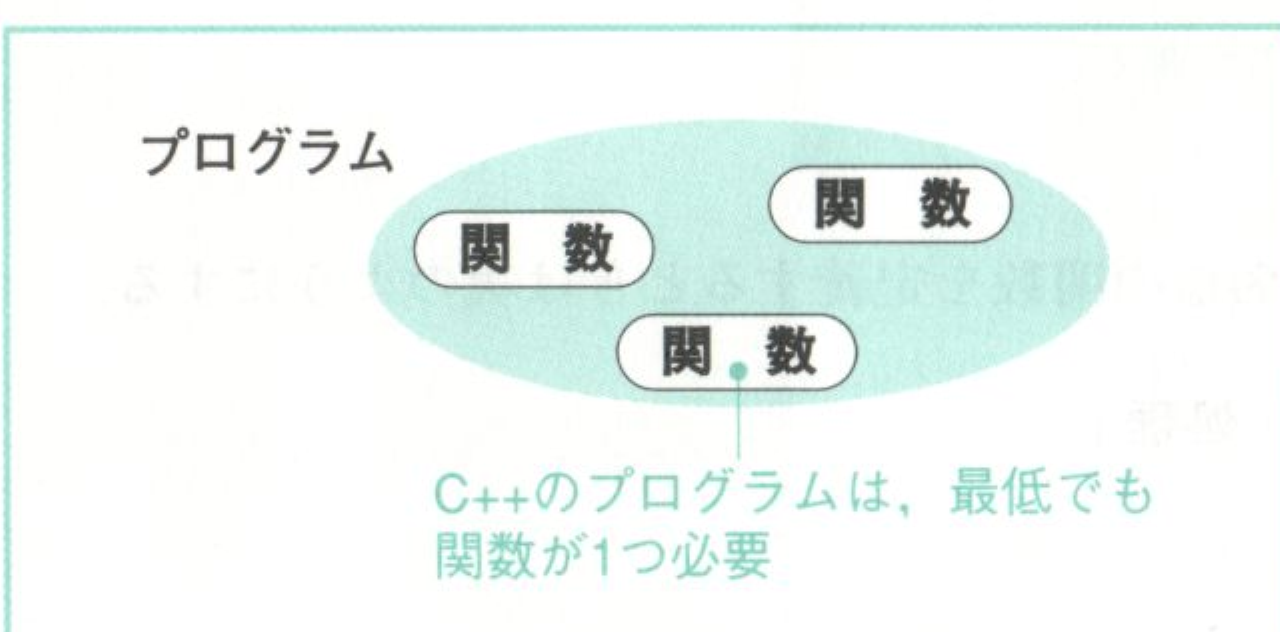
関数1
関数2
関数3
……

もちろん，関数が1つだけというプログラムもある．

関数◀――これだけでプログラムが完結している

ただし，関数が0個というのはダメである．C++のプログラムは，最低でも関数が1つ必要である．

図1-1▶



1.2◆関数

前項で「関数」という用語が出てきた．関数というと，数学を思い出すかもしれない．たとえば，次のような．

y = f(x)

これは「関数fにxという引数を与えると，何らかの変換（処理）を行って，yという値を返す」という意味である．C++の関数も，これに似ている．ただし，書き方が数学とはちょっと異なる．**C++の関数**は，次のように書く．

型　関数名(引数) { 処理 }

これだけではピンとこないだろう．次の節で，もう少し具体的にC++の関数の書き方を説明する．

2 関数の書き方

この節では，C++の関数の書き方を，その構成要素に分けて具体的に説明する．そして，実際に“funny”という名前の小さな関数を書き上げる．

2.1◆関数の名前

最初に，これから書こうとする関数の名前を考え，それを**関数名**のところに書く．

型　関数名(引数) { 処理 }

↑ ここに関数の名前を書く

たとえばfunnyという名前の関数を定義するときは次のようにする．

型　funny(引数) { 処理 }

↑ この関数の名前はfunny

関数の名前は，自由に付けることができる．ただし，次の規則に従う必要がある．

C++で使う名前

先頭は英字，または _（下線）のいずれかで始める

（数字で始まる名前を付けることはできない）

2文字目以後は英字，数字，_（下線）のいずれか

（2文字目以後は数字を使ってもよい）

最大の長さは決められていない

（C++コンパイラによって異なる）

大文字／小文字は区別される

注意　C++は，関数の名前だけではなく，いろいろな場面で「名前」を使う．たとえば後で出てくる変数にも名前を付ける．C++で使う名前は，上記の条件を満たしていなければならない．

たとえば，次は関数の名前として適当である．

funny
fn1 ――― 2文字目以後に数字があってもよい
_write ――― 先頭を下線で始めてもよい
long_name_100 ――― 途中に下線や数字があってもよい
a ――― 1文字だけでもよい

しかし，次はC++の関数名としては不適当である．

100pens ――― 数字で始めている
funny@ ――― 記号@は使えない
he is ――― 途中にスペースがある

大文字／小文字は区別されるので，次はいずれも異なる関数名である．

funny
Funny
fuNNy

C++の関数名は，小文字を使うことが多い．もちろん大文字を使っても間違いで

はない．長い単語を区切るときは，途中にスペースをはさむことはできないので，下線で区切る．

    this_is_a_pen

次のように大文字を使ってもよい．

    ThisIsAPen

この方式は，Pascal系出身の人に多い．

2.2◆引数のない関数

関数名の後ろに続く ( ) の中には，関数に与える引数を書く．

    型 名前(引数) { 処理 }

ここ

ただし，引数の書き方は（現段階では）ちょっと難しいので，もう少し後で説明する．ここでは，ただ ( ) だけを書いておく．( ) の中は空っぽのままにしておくのである．

    型 funny() { 処理 }

( ) の中は空のまま残す

このようにC++では，引数のない関数を作ることができる．そもそも関数は与えられた引数を何かに変換するものである．それゆえに「引数のない関数なんて変なの」と思うかもしれない．

引数のない関数は，値を返すのが目的ではなく，何らかの処理を行うことに意味を持つ関数である．C++で引数のない関数を作るには，( ) の中を空っぽにしておけばよい．いずれ引数を使うようになると，( ) の中にいろいろと記述するようになる．

注意　引数がないことを強調したいときは，( ) の中にvoidを書いてもよい．

    型 funny(void) { 処理 }

蛇足ながら，引数として空の ( ) を指定した場合，CとC++では意味が異なる．Cの場合，空の ( ) は引数の数が不定であることを示す．C++では，引数がないことを示す．C++で引数の数が不定であることを示すときは，ピリオドを3つ並べる．

2.3◆関数値

関数は，何かを処理して値を返す．関数が返す値を**関数値**という．

関数を定義するとき，いちばん最初に書く**型**というのは，関数が返す値——すなわち関数値の種類を示す．

型 funny() { 処理 }

↑

関数値の種類

C++の関数は，数値だけでなく，文字などいろいろな種類の値を返すことができる．その種類を示すのが，関数の先頭に書く**型**である．

ここでは，とりあえずintという型を覚える．intは，整数を表す**型名**（型の名前）である．

int funny() { 処理 }

↑

関数funnyの返す値が整数であることを示す

2.4◆return

funnyもだいぶ関数らしくなってきた．残るは，{ 処理 } の部分だけである．それでfunnyが完成する．

int funny() { 処理 }

↑

この部分を完成させる

関数定義の最後の部分——すなわち{}の中には，次の2つのことを書く．

- その関数が行う処理内容
- その関数が返す値（関数値）

ここでは，とりあえず関数値の書き方を覚えよう．

```
return 関数値;
```

returnの後ろにこの関数が返す値を書き，最後にセミコロンを付ける．たとえば0を返すなら，次のように書く（返す値は何でもよい）．

```
return 0;
```

この文を{}の中に書く．

```
int funny() { return 0; }
```

これで，最初の関数funnyが出来上がった．

3 コンパイル

関数funnyが出来上がった．この節では，それをC++のソースファイルにしてコンパイルするまでの手順を示す．

3.1◆ソースファイル

関数funnyは，次の1行からなる短いものである．

```
int funny() { return 0; }
```

適当なテキストエディタ（たとえばEmacsやMuleやviなど）で，この1行を入力したテキストファイルを作成する．ファイル名は“funny.C”とでもしておこう．

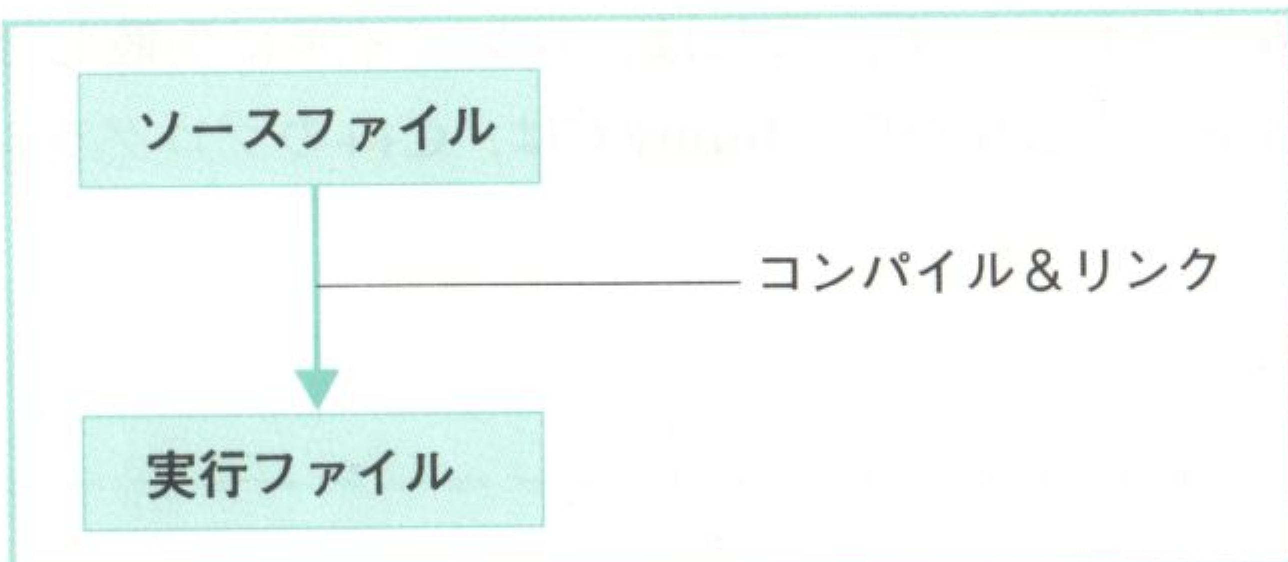
図1-2▼

```
% cat funny.C ———————— “funny.C”を作成する
int funny() { return 0; }
% _
```

注意　すぐ後で明らかになるが，残念ながら，このソースファイルはまだ完全ではない．

このようにC++のプログラムをテキストファイルにしたものを**ソースファイル**という．ソースファイルをC++コンパイラでコンパイルすることによって，実際に動くプログラム（**実行ファイル**）を作ることができる．

図1-3▶



C++のソースファイルを作るとき，ファイル名は（そのOSが許している範囲内なら）何でもよい．ただし，拡張子（ファイル名の最後の部分）は，使用するC++コンパイラによって決められている．

本書では，**GNU C++コンパイラ**を使用した．GNU C++コンパイラは，C++のソースファイルの拡張子として，次のいずれかを使うことができる．

.C
.cc
.cxx

別のC++コンパイラでは“.cpp”を使うように決めているかもしれない．

3.2◆コンパイル

ソースファイルを作成したらコンパイルする．コンパイルのしかたは，C++コンパイラによって異なる．GNU C++の場合は，次のようにする．

g++ ソースファイル名 Enter

注意　GNU C++コンパイラの使い方については，第4節で再び取り上げる．

次に，funny.Cをコンパイルしたところを示す．

図1-4▼

```
% g++ funny.C                           "funny.C" をコンパイルする
/usr/lib/crt1.o: In function `_start':
/usr/lib/crt1.o(.text+0x5a): undefined reference to `main'
% _
```

何やらメッセージが表示されている．これは，コンパイルに失敗したことを示す**エラーメッセージ**である．残念ながら，funny.Cは，C++のプログラムとして不完全である．

3.3◆関数main

コンパイラ（この場合はリンカ）が表示したエラーメッセージの意味は「このソースファイルには関数mainがないよ」といった内容のものである．

/usr/lib/crt1.o: In function `_start':
/usr/lib/crt1.o(.text+0x5a): undefined reference to `main'

注意　このエラーメッセージの内容をもっと詳しくいうと，「_startというスタートアップモジュールの中でmainという名前の関数を参照したが定義されていない」といった意味になる．

ここで，C++のプログラムについて新しい事項を覚える．本章第1.1節「C++のプログラム」で，C++のプログラムは関数を列挙したものであることを説明した．

関数1
関数2
関数3
.....

これをもう少し詳しくいうと，この中に1つ**main**という名前の関数がなければならない．

関数
関数
main ―――― どこかにmainという名前の関数が必要
関数
.....

さもないと，さきほどのコンパイルのようにエラーになってしまう．

C++のプログラムにはmainという名前の関数が必要である．その他の関数の名前は何でもよい．また，mainさえあれば，その他の関数はなくてもよい――これが，C++のプログラムが満たさなければならない最低の条件である．

3.4◆完成

それでは，funny.Cを修正してコンパイルできるようにしよう．新しくmainという名前の関数を追加してもよいが，手っ取り早いのは，関数funnyの名前をmainに変えてしまうことである．

```
int funny() { return 0; }
      ↓
int main() { return 0; }
```

新しいfunny.Cのソースファイルは次のようになる．

図1-5▼

```
% cat funny.C                    "funny.C" を作成する
int main() { return 0; }
% _
```

再びfunny.Cをコンパイルする．

図1-6▼

```
% g++ funny.C                    "funny.C" をコンパイルする
% _
```

今度はエラーメッセージが表示されることなく，コンパイルに成功している．このように，g++は，コンパイルに成功すると何も表示しないでプロンプトに戻る．

コンパイルが成功すると，**a.out**という名前の実行ファイルが作成される．

図1-7▼

```
% ls a.out
a.out ――――――――――――― 作成されている
% _
```

完成したプログラムを実行するには，コマンドラインで実行ファイルの名前（ここではa.out）をタイプする．

% a.out [Enter] （カレントディレクトリにパスが設定されていない場合は ./a.out [Enter]）

以下に，実行したところを示す．

図1-8▼

```
% a.out ――――――――――――――― 実行しても何も起こらない
% _
```

結果はご覧のとおりである．実行しても何も起こらない．これは当然である．何かが起こるべき処理を何も書いていないからである．何かが起こるプログラムについては，第3章までお預けとする．

4 GNU C++コンパイラ

第3.2節で GNU C++コンパイラによるコンパイルのしかたを説明した．この節は，その補足である．

4.1◆g++とgcc

funny.Cをコンパイルするのに，第3.2節では次のように入力した．

g++ funny.C Enter

これを見ると，あたかもg++というコマンドがC++コンパイラのように見える．しかし，実際は**gcc**というコマンドがfunny.Cをコンパイルしている．g++は，ただgccを呼び出しているに過ぎない．

gccは，GNUのコンパイラである．このコンパイラは高機能で，1つのコマンドで次の3つの言語をコンパイルすることができる．

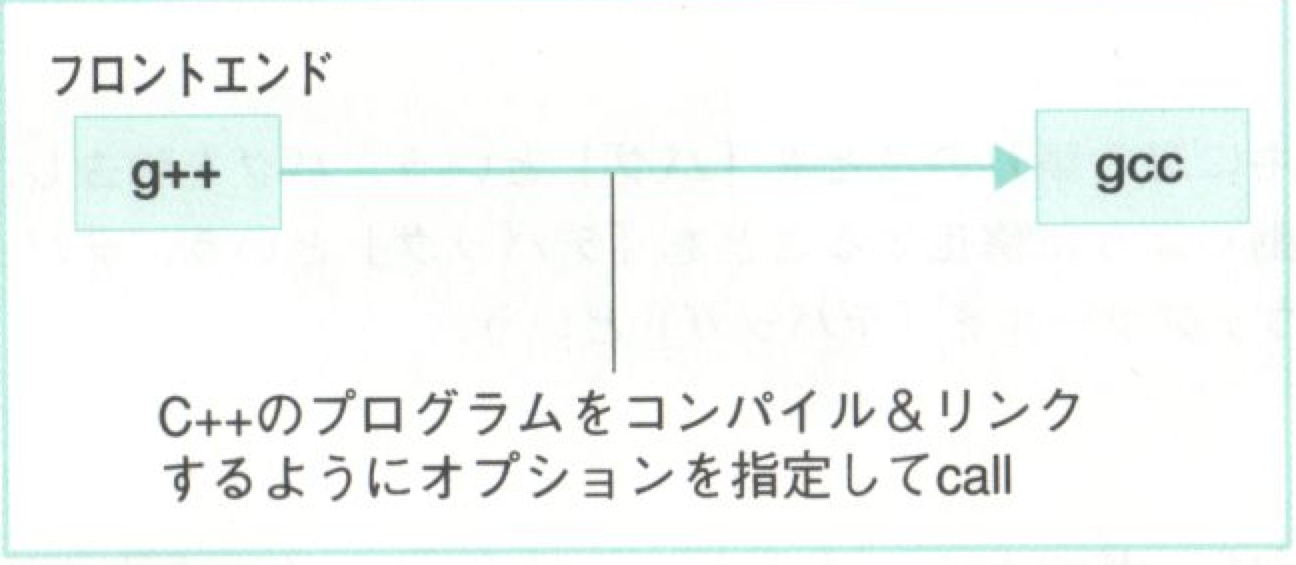
C
C++
Objective-C

gccをデフォルトで使うと（引数を指定しないで使うと），Cのプログラムをコンパイルしようとする．

gccでC++のプログラムをコンパイルする場合は，そのための引数を指定しなければならない．また，C++のライブラリをリンクするように指示しなければならない．これは，かなり面倒で，タイプも大変である．そこで，g++が用意された．

g++を実行すると，C++のプログラムをコンパイルするための引数を付けてgccを呼び出す．つまり，g++は，C++のプログラムを手軽にコンパイルするための，gccの**フロントエンド**というわけである．

図1-9▶



注意　ユーザーが直接実行する，あるいは対話するダミープログラムを「フロントエンドプロセッサ」という．あるいは，略して「フロントエンド」という．フロントエンドプロセッサの代わりに実際の処理を行うプログラムを「バックエンドプロセッサ」，あるいは略して「バックエンド」という．

フロントエンドプロセッサは，ユーザーとバックエンドプロセッサの仲介を行うプログラムということができる．

gccは，昔はシェルスクリプトであった．しかし，現在はバイナリになっている．

4.2◆実行ファイル名の指定

funny.Cをコンパイルするには，次のように入力する．

```
g++ funny.C
```

この結果，エラーがなければa.outという実行ファイルが作成される．しかし，いつも同じ実行ファイル名を使うのはまずいので，たいていは名前を変更する．

```
mv a.out funny
```
――実行ファイル名をfunnyに変更する

g++ に -o オプションを付けると，作成する実行ファイルの名前をa.outから任意の名前に変更することができる．たとえば次のようにしてコンパイルすると，（エラーがなければ）helloという名前の実行ファイルが作成される．このときa.outは作られない．

```
g++ -o hello funny.C
```

4.3◆デバッグを可能にする

GNU C++コンパイラで作成した実行ファイルを，同じくGNUから提供されている**gdb**というデバッガでデバッグすることができる．

補注　プログラム中に潜む誤りのことを「バグ」という．バグを除去し，プログラムが正しく動くように修正することを「デバッグ」という．デバッグに使われるソフトウェアツールを「デバッガ」という．

gdbは**ソースレベルデバッガ**である．すなわち，実行ファイルをあたかもソースファイルのようにしてデバッグすることができる．そのためには，実行ファイルの中からソースプログラムを参照できなければならない．つまり，gdbでデバッグするためには，ソースプログラムの参照を含む特別な実行ファイルが必要である．

gdbによるデバッグを行うためには -g オプションを指定して，次のようにコンパイルする．

```
g++ -g funny.C
```

このようにしてコンパイルされた実行ファイルには，デバッグに必要な情報

(たとえばソースファイルへの参照) が含まれる.

なお, デバッガを使うときは, -gといっしょに-O0 (マイナス・大文字のオー・ゼロ) というオプションもいっしょに指定するとよい.

g++ -g -O0 funny.C

このオプションを付けないと, デバッグ中, 思わぬハプニングに出遭うことがある.

GNU C++コンパイラは賢い. ソースファイル中に含まれる無駄なコードを除去し, 速く, しかもできるだけ小さな実行ファイルを作成しようとする. これを実行ファイルの**最適化**という.

最適化を行うと, ソースファイルの1行1行が, 実行ファイルと対応しなくなる. これは, デバッグの妨げとなる. -O0 (マイナス・大文字のオー・ゼロ) オプションを付けると, 最適化を行わなくなる. デバッグ用の実行ファイルを作成するときは, -gオプションとともにこのオプションを指定して, 最適化をしないようにすべきである.

以下に, funny.Cをgdbでデバッグしているところを示す.

図1-10▼

```
% g++ -g -O0 go.C ――――――――――――― デバッグ用の実行ファイルを作成する
% gdb a.out ――――――――――――――――― デバッグの開始
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i586-unknown-linux),
Copyright 1996 Free Software Foundation, Inc...
(gdb) list ――――――――――――――――― ソースコードを表示する
1       int main() { return 0; }
(gdb) run ―――――――――――――――――― プログラムを実行する
Starting program: /home/kt/a.out

Program exited normally.
(gdb) quit ――――――――――――――――― デバッガを終了する
% _
```

-gや-O0オプションを付けてコンパイルすると, (デバッグ情報が含まれるし, 最適化が行われないので) 実行ファイルは大きくなる. デバッグが終了し, プログラムが完成したら, これらのオプションを付けないでコンパイルし直して完成版の実行ファイルを作成すべきである.

第2章 書式の慣例

コンパイル可能な（しかし，何も起こらない）C++のプログラムを書くところまで進んだ．次のステップに進む前に，この章でC++のプログラム全体に関わる事項を説明しておく．

1 トークン

C++のプログラムをその構成要素に分解していくと，それ以上は分解できないというギリギリの単位が得られる．それが**トークン**（*token*）である．つまり，C++のプログラムを構成する1つ1つの単語がトークンである．

1.1◆トークン

前の章で取り上げたfunny.Cを例に説明する．funny.Cは，次の1行からなる短いものであった．

```
int main() { return 0; }
```

これをトークンに分解すると，次のようになる．

```
int
main
(
)
```

```
{
return
0
;
}
```

つまり，funny.Cは全部で9個のトークンで構成されていることになる．

1.2◆トークンの分解

トークンは，いわばC++プログラムの最小単位といえる．トークンをそれ以上に分解すると，もはやC++から認識することはできなくなってしまう．したがって，トークンの途中にスペースを入れたり，トークンの途中で改行したりすることはできない．たとえば，returnというトークンを次のように書いてはいけない．

```
ret urn ―――――― トークンの途中にスペースが入っている

re
turn ―――――― トークンの途中で改行している
```

1.3◆トークンとトークン

原則として，トークンとトークンをくっつけて書いてはいけない．トークンの間には，1つ以上の空白を置かなければならない．**空白**とは，スペースかタブか改行のことである．

```
int    main ―――――― トークンとトークンの間を
                    4つの空白で区切っているので正しい
intmain ―――――― トークンとトークンをくっつけているので誤り
                    C++はintmainを1つの単語として認識してしまう
```

しかし，( ) { } ; といった**区切り記号**なら，他のトークンとくっつけて書いてもかまわない．

```
main  (  ) ―――――― mainと"("と")"をそれぞれ離している
                    正しい
main() ―――――― mainと"("と")"をすべてくっつけている
                    これも正しい
```

2 自由書式

世の中にはたくさんのプログラミング言語が存在する．規律の厳しいもの，緩いものなど，さまざまある．

たとえば，BASICのプログラムは行単位で記述しなければならない．1つの文を複数の行にまたがって書くことはできない．また，FORTRANになるともっと厳しくて，行どころか1行の何桁目に何を書くかということまで事細かく決められている．

それらの言語に比べると，C++は比較的規律の緩やかな言語である．1つの行にいくつの文を書いてもよいし，逆に1つの文を複数の行に分けて書くこともできる．

```
return 0; return 0;        ――1つの行に複数の文を書いた

return                     ――1つの文を3行に分けて書いた
0
;
```

それどころか，C++には「行」の概念が存在しない．存在するのは「トークン」の概念だけである．1つのトークンを途中で区切ることさえしなければ，トークンとトークンの間をどれだけ空けようが，改行しようが自由である．たとえば，同じfunny.Cでも，次のようにいろいろな形式で書くことができる．

```
int funny() { return 0; }  ――すべてを1行に詰め込んだfunny.C

int                        ――3行に分けて書いたfunny.C
funny (     ) {
return 0    ; }
```

どちらも同じようにコンパイルすることができる．

C++のように自由な書き方を許す言語を**自由書式**（フリーフォーマット；*free-format*）な言語という．

3 書式の習慣

C++は，フリーフォーマットな言語である．とはいうものの，各自がそれぞれバラバラな書式でプログラムを書くと，読みにくいプログラムになってしまう．そこで，ある程度プログラムの書き方についての習慣が決められている．

3.1◆1行1文

まず，原則として，1つの行には1つの文を置く．if文のように1つの文を複数の行に分けて書くことはあるが，複数の文を1つの行に置くことはない．

注意　　if文は，第5章「選択文」のところで取り上げる．

3.2◆関数の書式

関数については，次のような書き方をする．

```
型　関数名(引数)
{
	文
	文
	…
}
```

たとえば，関数funnyなら次のように書く．

```
int funny()
{
	return 0;
}
```

人によっては，{ を（ ）の後ろに書く人もいる．

```
int funny() {  ←── ここ
	return 0;
}
```

どちらの書き方でもよいが，本書ではC++の開発者**ストラウストラップ**（Bjarne Stroustrup）風の書き方を踏襲する．すなわち，前者の書き方をする．

3.3◆インデント

C++の文には，書式上のレベルがある．各レベルをわかりやすくするため，下位のレベルを右側に段付けして書く．

```
aaaaaa  ―――――― 最上位のレベルの文
aaaaaa
    bbbbb ―――― 1つ低いレベルの文
    bbbbb
        cccccc ―― もう1つ低いレベルの文
        cccccc
    bbbbb
aaaaa
```

これを**インデント**（*indent*）という．インデントに何文字使うかは，言語によって習慣が違う．たとえば，Pascalは2文字ずつインデントする．また，CやC++の場合は，2文字，4文字または8文字である．本書では，4文字ずつインデントする．

どのような場合にインデントするかは，新しい文が出てくるたびにパターンを示す．そこで，まず覚えてほしいのが，関数の内部に書く文である．関数の { と } の内部に書く文はインデントする．つまり，次のようには書かない．

```
int main()
{
return 0;  ◀―――― インデントされていない
}
```

次のように書く．

```
int main()
{
    return 0;  ◀―――― インデントしている
}
```

4 コメント

C++のソースファイルの中には**コメント**（*comment*）を書くことができる．すなわち，プログラムにメモを残すことができる．コメントは，コンパイラのプリプロセッサ（前処理プログラム）により除去され，1つのスペースに変換される．したがって，コメントがプログラムの実行に影響を与えることはない．

プログラムについての説明やメモなどをコメントにして残しておくと，後日プログラムを読むときの参考になる．

4.1◆C++のコメント

C++では，次の2種類のコメントを併用できる．

- 行形式（C++形式）

  //から行末までがコメントとして扱われる．

- ブロック形式（C形式）

  /*から*/の間がコメントとして扱われる．
  複数行にわたってコメントを記述できる．

4.2◆行形式のコメント

一般的には，2つのスラッシュを重ねた//を使う．//から行末までがコメントになる．たとえば，次はfunny.Cにコメントを追加したものである．

```
//
// funny.C -- C++の最小プログラム
//

// 次は関数mainの定義
int main()
{
    return 0;    // 関数mainの関数値を定義している部分
}
```

このように//を行の先頭から始めてもよいし，行の途中から始めてもよい．

4.3◆ブロック形式のコメント

C++はブロック形式のコメント――すなわち，Cと同じ形式のコメントを使うこともできる．この形式では，/*と*/にはさまれた部分がコメントになる．第4.2節のコメントをブロック形式で書くと，次のようになる．

```
/*
   funny.C -- C++の最小プログラム
*/

/* 次は関数mainの定義 */
int main()
{
     return 0;    /* 関数mainの関数値を定義している部分 */
}
```

ブロック形式のコメントは，複数の行にまたがることができる．

4.4◆コメントのネスト

コメントを含む部分をコメントにすることをコメントの**ネスト**（*nest*）という．行形式のコメントとブロック形式のコメントを組み合わせれば，コメントのネストが可能になる．たとえば，次のようなプログラムがあったとする．

```
// 次は関数funnyの定義
int funny()
{
     return 0;    // 関数funnyの関数値を定義している部分
}

// 次は関数mainの定義
int main()
{
     return 0;    // 関数mainの関数値を定義している部分
```

```
}
```

ここで，別の種類のコメントを使って関数funnyの部分を一時的に削除することができる．

```
/*  ←――――――――――― ここから
// 次は関数funnyの定義
int funny()
{
    return 0;   // 関数funnyの関数値を定義している部分
}
*/  ←――――――――――― ここまでをコメントにする

// 次は関数mainの定義
int main()
{
    return 0;   // 関数mainの関数値を定義している部分
}
```

このように，いつもは行形式のコメントを使ってプログラムを書く．そして，デバッグ中にある部分を一時的に削除したいときは，その部分をブロック形式のコメントにすればよい．

第3章

ストリームI/O

これまでC++のプログラムとして最小限備えていなければならないプログラム要素について説明してきた．そして，コンパイル可能なfunny.Cというプログラムを作成した．しかしながら，funny.Cを実行しても，何も起こらなかった．これでは面白くも何ともないので，この章から**何かが起こる**プログラムを作成していく．

1 cout

「プログラムが動いた！」という感動を手っ取り早く得るには，ディスプレイに何か文字列を表示してみることである．

> **注意**　しばらく「表示する」とか「出力する」といった用語を用いるが，第5章でもう一度これらの用語について再検討する．

1.1◆文字列の表示

C++で文字列を表示するには，次のようにする．

```
cout << "文字列";
```

たとえば，次は“Hello, world”という文字列を表示する文である．

```
cout << "Hello, world";
```

この文をプログラムの中の「どこか？」に書けば，“Hello, world”を表示することができる．それでは，この文をプログラムの中の「どこに」書けばよいだろう？

1.2◆実行の流れ

ここで，C++におけるプログラムの「実行の流れ」をまとめておこう．C++のプログラムは，次のように実行される．

C++の実行の流れ

1. 関数mainの最初の文から実行が開始される
2. 2番目以後の文が順に実行される
3. 次のいずれかで実行が終了する
   - 関数mainの最後の文に到達した
   - 関数mainの中でreturn文に到達した
   - 関数exitが実行された

注意　関数exitについては，この章では説明しない．

たとえば，次のようなプログラムがあったとする．

```
int main()  ―――― 関数main
{
    文1;  ―――― 最初の文
    文2;  ―――― 2番目の文
    文3;
}
```

この場合，プログラムは「文1→文2→文3」のように実行される．また，次のように，途中にreturn文があったとする．

```
int main()
{
    文1;
    文2;
```

```
        retunr 0;
        文3; ──────────── この文は実行されない
    }
```

この場合，プログラムは「**文1→文2→return文**」のように実行される．**文3**は実行されずに取り残される．

return文がmainの途中にあると，そこでプログラムが終了してしまう．通常，return文は関数の最後に書く．あるいは，後の章で出てくる選択文の中で使われる．

1.3◆coutの位置

これでcoutをどこに置けばよいかが決まった．関数mainの中でreturn文よりも前の位置に置く．さもないと，実行されずに取り残されてしまう．したがって，次のようなプログラムを作れば，ディスプレイに文字列を表示できる（はずである）．

【サンプルプログラム】

これはcoutを使って文字列の表示を試みるプログラムである．正しいプログラムかどうかは，まだわからない．

```
// hello.C -- "Hello, world"を表示する

int main()
{
    cout << "Hello, world"; ──────── この位置に注意
    return 0;
}
```

さっそくhello.Cをコンパイルして実行してみよう……といいたいところであるが，残念ながら，このプログラムをコンパイルするとエラーになる．

【g++によるコンパイル例】

```
% g++ hello.C
hello.C: In function `int main()':
hello.C:5: `cout' undeclared (first use this function)
hello.C:5: (Each undeclared identifier is reported only once
hello.C:5: for each function it appears in.)
% _
```

ここに表示されているメッセージは，g++コンパイラによるエラーメッセージである．このエラーの意味は，「関数mainの中で使っているcoutが定義されていないよ」といった内容のものである．これについては，節を改めて説明することにする．

2 ストリームI/O

coutを使ったプログラムがコンパイルエラーになってしまった．それにしても，そもそもcoutとは何だろう？

returnと同じようにC++に用意された機能の一部だろうか？——答えは否である．

2.1◆coutの正体

C++から見ると，returnとcoutはまったくの別物である．returnはあらかじめC++に組み込まれたC++の機能の1つである．したがって，C++はreturnのことをよく知っている．

一方，coutはC++の機能の1つではない．したがって，C++はcoutのことをまったく知らない．そのため，ソースファイルの中にcoutが出てきても，それをどのように処理すればよいのかわからない．coutを使ったプログラムがコンパイルエラーになったのは，このためである．

coutを（コンパイルエラーにならないように）C++に認識してもらうためには，自己紹介が必要である．coutがC++に対して挨拶（あいさつ）する必要がある．以下，その方法を説明する．

2.2◆ストリームI/O

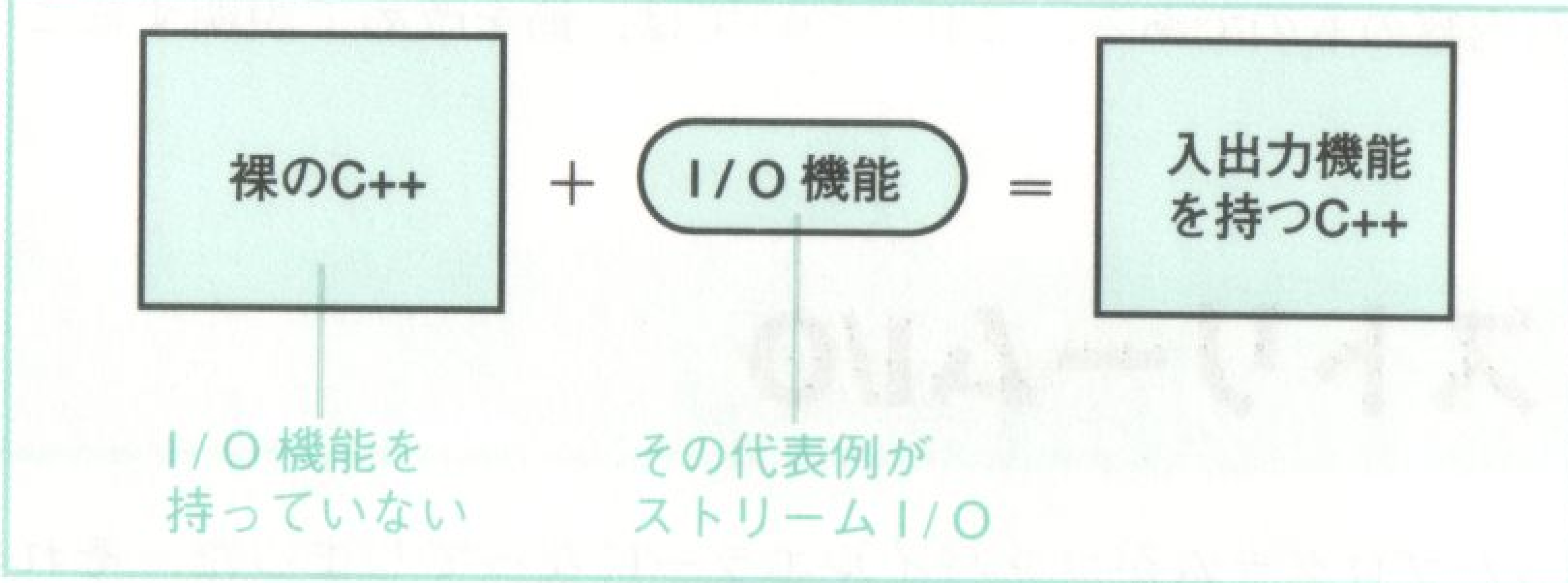
C++は，C言語と同じように**I/O機能**（入出力機能）を言語仕様としては持っていない．つまり，裸のままのC++では，ディスプレイに文字列を表示したり，キーボードからデータを入力したりすることはできない．

これでは言語として使い物にならない．しかし，各自がC++の標準機能を使ってI/O機能を追加することはできる．つまり，各自が必要なI/O機能を追加するのである．とはいうものの，それは簡単な作業ではない．

そこで，C++を開発したストラウストラップが**ストリームI/O**というI/O機能を開発した．ストリームI/Oなら，たいていのC++処理系に付属している．これを利

用することにより，自分でプログラムすることなくI/O機能を使うことができる．

図3-1▶



2.3◆iostream.h

ストリームI/Oを利用するためには，プログラムの先頭部分に次の1行を入れる必要がある．

```
#include <iostream.h>
```

これが，coutをC++に認識してもらうための**おまじない**である．

注意
1. coutはC++のclassや演算子の多重定義を使って巧妙に作られている．
2. ストリームI/Oの詳細については，拙著『ストリームI/Oの謎』（星雲社発行）を参照されたい．ただし，現在では入手が難しいかもしれない．

2.4◆Hello, world

第1.3節で示したhello.Cをコンパイルできるようにするには，プログラムの先頭部分に次の1行を追加すればよい．

```
#include <iostream.h>
```

次に改良版のhello.Cを示す．

【サンプルプログラム】

これは，第1.3節のプログラムをコンパイルエラーが出ないように改良したものである．

```
// hello.C(第2版) -- "Hello, world"を表示する

#include <iostream.h>  ―― この行を追加する

int main()
{
    cout << "Hello, world";
    return 0;
}
```

次に，この改良版をコンパイルしたところを示す．

【g++によるコンパイル例】

```
% g++ funny.C  ―― "funny.C" をコンパイルする
% _
```

今度はエラーメッセージが表示されることなく，コンパイルに成功している．次に実行例を示す．

【実行例】

```
% a.out
Hello, world% _
```

"Hello, world" の文字列が表示されている．成功である．しかし，ちょっと変である．"Hello, world" の行と次のプロンプトが同じ行に表示されている．見にくい！

この問題を次の「マニピュレータ」の節で解決しよう．

3 マニピュレータ

この節では，マニピュレータを使って強制的に改行させる方法を説明する．マニピュレータは，ストリームI/Oの一機能である．

3.1◆coutによる出力

第2節のサンプルプログラムでは，coutによる出力が次のプロンプトとくっついてしまった．

Hello, world%

↑

ここの部分

この理由を考えるため，次のプログラムをコンパイルして実行する．

【サンプルプログラム】

これは，coutによる改行をテストするものである．

```
// lines.C -- 複数行の出力

#include <iostream.h>

int main()
{
    cout << "Where did you find it ?";  ―― 1行目
    cout << "It was on the floor.";  ―― 2行目
    return 0;
}
```

【実行例】

```
% a.out
Where did you find it ?It was on the floor.% _
```

このプログラムはcoutを使って2行分の文字列を出力しようとしたものである．結果は，期待に反したものになっている．すなわち，その2行が1行にまとめられて出力されている．もちろん，次のプロンプトもくっついてしまっている．

この実行結果から，「coutによる出力は（自動的には）改行しない」ことがわかる．

重要　coutによる出力は（自動的には）改行しない!!

複数の行を表示したり，プログラム終了時に次のプロンプトと分離するには，明示的に改行を指示する必要がある．

3.2◆マニピュレータ

coutによる出力を改行させるには，次の2つの方法がある．

- エスケープシーケンスの\nを使う
- マニピュレータのendlを使う

ここでは，**マニピュレータ**を使う方法を説明する．マニピュレータというのは，ストリームI/Oの中で次に示す記号の右側に置くことのできる特別な関数である．

<< ――データを右辺から左辺に流す
（右辺のデータを左辺に出力する）

>> ――データを左辺から右辺に流す
（左辺から入力したデータを右辺に代入する）

そのマニピュレータの1つに**endl**というのがある．endlを>>の右側に置くことによって強制的に改行させることができる．たとえば，“Hello, world” を出力後，改行させるには，次のようにする．

```
cout << "Hello, world";
cout << endl;  ――――――――ここで改行する
```

これで第2.4節のhello.Cを改良することができる．

【サンプルプログラム】

これは，マニピュレータのendlをテストするものである．

```
// hello.C(第3版) -- "Hello, world"を表示する
#include <iostream.h>

int main()
{
    cout << "Hello, world";
    cout << endl;  ――――――――――――――この行を追加する
    return 0;
}
```

【実行例】

```
% a.out
Hello, world
% _ ◀──────────────── 離れた！
```

これで“Hello, world”とプロンプトがくっつかなくなった．第2.4節の問題は解決した．しかし，これをもう少しエレガントに書く方法がある．

3.3◆<<の連鎖

複数のcoutを書くとき，<<を連鎖させて1つのcoutにまとめることができる．たとえば，

```
cout << "Hello, world";
cout << endl;
```

のような2行を，次の1行にまとめることができる．

```
cout << "Hello, world" << endl;
```

また，第3.1節のlines.Cを1行ずつ改行させるには，基本的は次のように書く．

```
cout << "Where did you find it ?";
cout << endl;
cout << "It was on the floor.";
cout << endl;
```

これをまとめて，次のように書くこともできる．

```
cout << "Where did you find it ?" << endl
     << "It was on the floor." << endl;
```

読みやすくするため2行に分けて書いてあるが，1つの文である．coutも1つしか使っていない．4つの<<を連鎖させた結果である．

次に，hello.Cの最終版と実行例を示す．

【サンプルプログラム】

これは，hello.Cを，<<を連鎖させて書いたものである．これをもってhello.Cの

改良も打ち止めとする.

```
// hello.C(最終版) -- "Hello, world"を表示する

#include <iostream.h>

int main()
{
    cout << "Hello, world" << endl;
    return 0;
}
```

【実行例】

```
% a.out
Hello, world
% _
```

複数の行を出力するlines.Cの最終版と実行例を次に示す.

【サンプルプログラム】

これは，lines.Cを，<<を連鎖させて書いたものである.

```
// lines.C(最終版) -- 複数行の出力

#include <iostream.h>

int main()
{
    cout << "Where did you find it ?" << endl
         << "It was on the floor." << endl;
    return 0;
}
```

【実行例】

```
% a.out
Where did you find it ?
It was on the floor.
% _
```

それぞれの行がきちんと改行され，プロンプトも独立した行に表示されるようになっている.

第4章 演算とデータ型

この章では，数についての演算，およびデータ型について説明する．coutによる出力も一挙にバラエティが増える．

1 数値の出力

これまでcoutを使って文字列を表示してきた．しかし，coutは文字列を表示するためだけにあるのではない．数値を出力することもできる．

注意　この節では「数値」という言葉を使用するが，これはあまり正確な用語ではない．C++の数値には，いくつかの種類がある．これについては第3節で説明する．

1.1◆数値の出力

数値を出力するやり方は，文字列のときと同じである．演算子<<の左側にcoutを置き，<<の右側に出力するデータを置く．

```
cout << "……";     ———— 文字列を表示する（文字列は "" で囲む）
cout <<数値；     ———— 数値を表示する（数値は "" で囲まない）
```

次に例を示す．

【サンプルプログラム】

これはcoutを使って数の1000を出力するものである．

```
// num.C -- 数値を出力する

#include <iostream.h>

int main()
{
    cout << 1000 << endl;   ―――― この行で数値を表示
    return 0;
}
```

【実行例】

```
% a.out
1000 ―――― 数値が表示されている
% _
```

注意　数値を表示するときにも，endlを使わないと，プログラム終了時に次のプロンプトとくっついてしまう．

さて，この出力を見るかぎり，文字列を表示しても数値を表示しても，あまり違いがないように見える．

```
cout << 1000 << endl;   ―――― 数値の1000を表示する
cout << "1000" << endl; ―――― 文字列の1000を表示する
```

この両者は同じ1000を表示する．結果的には同じである．では，わざわざ数値を使うのはなぜだろう？

1.2◆演算

文字列の代わりに数値を使うのは，数値なら**演算**が可能だからである．たとえば，次は1200を表示する．

cout << 1000 + 200 << endl;

【サンプルプログラム】

これは数値を使って演算を行い，その結果を出力するプログラムである．

```
// num.C(第2版) -- 数値を出力する

#include <iostream.h>

int main()
{
    cout << 1000 + 200 << endl;
    return 0;
}
```

【実行例】

```
% a.out
1200 ――――――― 1000 + 200が計算され
                その結果が表示される
% _
```

このようにcoutの<<の右側には数値の代わりに**式**を置くこともできる．式を置くと，その式が評価され（計算され）その結果が出力される．

同じことを文字列で行うことはできない．たとえば，次のような文はコンパイルエラーになる．

cout << "1000" + "200" << endl;

注意　文字列については「**連結**」という演算を行うことができる．これは2つの文字列をつなげて1つの文字列にする演算である．

文字列を連結するには，2つの文字列を並べればよい．文字列と文字列の間には演算子を置かない．たとえば "Hello, " "world" は連結されて "Hello, world" になる．

2 四則演算子

数値について**四則演算**を行うには，次の演算子を使う．

演算子	演算
+	加算
-	減算
*	乗算
/	除算
%	剰余

2.1◆数の合計

ここでは，例として数の合計を計算するサンプルプログラムを2つ紹介する．

【サンプルプログラム】

これは，1～10までの合計を表示するサンプルプログラムである．

```
// count10.C -- 1～10までの合計を表示する

#include <iostream.h>

int main()
{
      cout << 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 +10
              << endl;
     return 0;
}
```

【実行例】

```
% a.out
55 ――――――――――――――――――― 1～10までの合計が計算され
                                        その結果が表示される
% _
```

同じく，次は1～1000までの合計を表示する．

【サンプルプログラム】

```
// count1000.C -- 1～1000までの合計を表示する

#include <iostream.h>

int main()
{
    cout << (1 + 1000) * 1000 / 2 << endl;
    return 0;
}
```

【実行例】

```
% a.out
500500 ―――――――――――― 1～1000までの合計が計算され
                       その結果が表示される
% _
```

ここで * は×（掛ける）を意味する演算子である．キーボードには×の記号がないので，代わりに * を使用する．また / は÷（割る）を意味する演算子である．キーボードには÷の記号がないので，代わりに / を使用する．

この例では，演算子の**優先順位**を変えるため（）も使われている．加減よりも乗除が優先されるので，（）がなければ正しい答えが得られない．

2.2◆元利合計

ここでは，元利合計を計算するプログラムを紹介する．

【サンプルプログラム】

これは，1000円を年利5%の利率で預けた場合の1年後の元利合計（元金＋利息）を計算するプログラムである．

```
// amount.C -- 元利合計を計算する

#include <iostream.h>

int main()
{
    cout << 1000 * 1.05 << endl;
    return 0;
}
```

【実行例】

```
% a.out
1050
% _
```

1050 ―― 元利合計が計算され その結果が表示される

2.3◆オリンピックの開催

算術演算子の%は，整数の除算において**剰余**（いわゆる余り）を計算する．例として，オリンピックが開催される年かどうかを計算するサンプルプログラムを次に示す．

オリンピックは，4年に1回，4で割り切れる年に行われる．ある年（たとえば1998年）にオリンピックが開催されるかどうかを調べるには，次の文を実行してみればよい．

ここに調べたい西暦を書く（→ 1998）

```
cout << 1998 % 4 << endl;
```

この出力が0になれば，その年にはオリンピックが開催される．

【サンプルプログラム】

```
// olimpic.C -- オリンピックの開催を調べる

#include <iostream.h>

int main()
{
    cout << 1998 % 4 << endl;
    return 0;
}
```

【実行例】

```
% a.out
2
% _
```

2 ―― 4で割って余りが2になっているので 1998年はオリンピックの年ではない

3 いろいろなデータ型

第1節で「数値」という言葉を使用したが，これはあまり正確な用語ではない．C++で使われるデータには，「整数」とか「浮動小数点数」とか「文字列」とかのような，いろいろな種類がある．これを**データ型**という．

この節では，C++で使われる基本的なデータ型，およびその表現方法について説明する．

3.1◆データ型

以下に，C++で使われる基本的なデータ型をいくつか示す．

データ型	意味
文字	1バイトの1文字を表す 漢字は2バイトなので文字型では表現できない
文字列	文字の並び 0文字でも，1文字でも，2文字以上でもよい 処理系によっては漢字を含めることもできる
整数	小数点を含まない数 8進数や16進数で表現することもできる
浮動小数点数	小数点を含む数

ひと口に「数値」といっても，小数点以下があるかないかで，整数と浮動小数点数の2種類に分かれる．

3.2◆文字／文字列リテラル

“Hello, world”とか100とか3.14のような個々の具体的なデータを**リテラル**（*literal*）という．データ型により，リテラルの書き方は異なる．

●文字リテラル

文字型のリテラルを表すには，その文字を単一引用符（'）で囲む．次に例を示す．

'A'―― 大文字のAを表す文字リテラル

'z'—— 小文字のzを表す文字リテラル
'@'—— 記号の@を表す文字リテラル
'漢'—— たとえ1文字でも漢字はダメ
文字リテラルにならない

●文字列リテラル

文字列のリテラルについては，すでにプログラムの中でいくつか使っている．文字列を二重引用符（"）で囲めばよい．次に例を示す．

"Hello, world"
"Where did you find it ?"
"It was on the floor."
"漢字"—————— 処理系によっては漢字を含めることができる
""—————— 空の文字列

3.3◆数値リテラル

数値リテラルには，**整数リテラル**と**浮動小数点数リテラル**の2種類がある．

●整数リテラル

整数を表すリテラルには，次の3種類がある．

種類	書き方
10進数リテラル	0（ゼロ）で始まらない整数
8進数リテラル	0（ゼロ）で始まる8進数 ただし，0の後ろにxやXが続かないこと 8や9を使ってはいけない
16進数リテラル	0xまたは0Xで始まる16進数 10～15までの数はa～fまたはA～Fを使う

次に例を示す．

100 —————— 10進数リテラル
0777 —————— 8進数リテラル
0819 —————— エラー

0で始めているので8進数として解釈されるが，8や9は8進数ではない

0x12fe ―― 16進数リテラル

●浮動小数点数リテラル

実数を表現するには，浮動小数点数リテラルを使う．これは，次のように小数点以下を追加すればよい．

3 ―― これは整数リテラル

3.14159 ―― これは浮動小数点数リテラル

小数点以下がない数でも浮動小数点数リテラルにするには，必ず小数点を付ける．たとえば，次はどちらも3という値を持つが，前者は整数，後者は浮動小数点数になる．

3 ―― これは整数リテラル

3.0 ―― これは浮動小数点数リテラル

小数点以下の数がない場合，ピリオドの右側を省略してもよい．

3 ―― これは整数リテラル

3. ―― ピリオドが付いているので，これは浮動小数点数リテラル

逆に整数部分が0の場合は，ピリオドの左側を省略してもよい．たとえば，次はいずれも同じ値を持つ浮動小数点数である．

0.1234

.1234

なお，第6章第2.3節で説明する予定であるが，浮動小数点数には表現可能なサイズによって次の3種類がある．

float

double

long double

浮動小数点数リテラルは，特に明示しないかぎりdoubleとなる．floatであることを明示するためには，最後にfまたはFを付ける．また，long doubleであること

を明示するためには，最後にl（小文字のエル）またはLを付ける．

3.14159 ―― これはdouble型の浮動小数点数
3.14159f ―― これはfloat型の浮動小数点数
3.14159L ―― これはlong double型の浮動小数点数

第5章 標準出力

この章では，coutによる出力を標準出力の観点から見直す．

1 いろいろな出力

前の章で，文字型とか整数型とか浮動小数点型とか，いろいろな型のデータを見てきた．coutは，これらの基本型をすべて出力することができる．coutを<<の左側に，また<<の右側に出力したいデータを置けばよい．

cout <<いろいろな型のデータ；

次に例を示す．

```
cout << 'A' << endl;                 文字の出力
cout << "Hello, world" << endl;      文字列の出力
cout << 512 << endl;                 整数の出力
cout << 2.2360679 << endl;           浮動小数点数の出力
```

異なるデータ型を一度に出力することもできる．

```
cout << "円周率 = " << 3.14159 << endl;
```

"円周率 = " ← 文字列　　3.14159 ← 浮動小数点数

2 もう1つの視点から

coutによる出力をいろいろ説明してきた．ここで別の角度からcoutを見直す．

2.1◆標準出力

これまでcoutによる出力を「ディスプレイに表示する」とか，単に「出力する」とかのように曖昧(あいまい)ないい方をしてきた．もう少し正確にいい直すと，coutはデータを**標準出力**に出力する．これは，必要ならcoutの出力先を任意のファイルに**リダイレクト**できることを意味する．

注意　リダイレクトは，標準出力の出力先を指定のファイルへ切り替える．詳しくは，UNIXの教科書を参照されたい．

第3章「ストリームI/O」で取り上げたhello.Cを例に説明する．標準出力は，デフォルトでは端末（たとえば，X上ならkterm）に結び付けられている．つまり，普通に実行すると，端末に出力される．

図5-1▼

```
% a.out
Hello, world ──────── 端末に出力される
% _
```

このような使い方では，「ディスプレイに表示する」という意味もあながち間違いではない．ただし，X上では，むやみやたらなところに表示されるわけではなく，ktermなどの端末エミュレータに表示されるので「端末に出力する」というほうがより正確である．

2.2◆リダイレクト

同じプログラムを再コンパイルすることなく，任意のファイルにリダイレクト

することもできる．たとえば，次はa.outによる出力をhello.outというファイルにリダイレクトしている．

図5-2▼

```
% ls hello.out                                  hello.outは存在しない
ls: hello.out: No such file or directory
% a.out > hello.out                             hello.outにリダイレクト
% ls hello.out
hello.out                                       hello.outが作成されている
% cat hello.out
Hello, world                                    coutによる出力
% _
```

リダイレクトすると，coutによる出力が端末には出力されないことに注意．

```
% a.out > hello.out     hello.outにリダイレクト
% _                     端末には出力されず
                        すぐにプロンプトに戻っている
```

代わりにhello.outというファイルが作成され，coutによる出力はそのファイルに対して行われる．

coutによる出力をプリンタに行うには，次のようにする．

図5-3▼

```
% a.out > /dev/lp1                  プリンタにリダイレクト
% _
```

注意　上のようにして印刷できるためには，/dev/lp1がプリンタのデバイスファイルであり，かつプリンタ関係の設定が正しく行われていて，かつユーザーに対するパーミッションが有効でなければならない．

2.3◆teeコマンド

端末と任意のファイルに同時に出力するには，teeコマンドを使う．

図5-4▼

```
% ls foo ―――――――――――――――――――― ファイルfooは存在しない
ls: foo: No such file or directory
% a.out | tee foo ――――――――――――― パイプでteeコマンドにつなぐ
Hello, world ――――――――――――――――― 端末に出力されている
% ls foo
foo ――――――――――――――――――――――― ファイルfooが作成されている
% cat foo
Hello, world ――――――――――――――――― ファイルfooにも出力されている
% _
```

2.4◆/dev/null

/dev/nullにリダイレクトすることにより，出力を捨てることもできる．

図5-5▼

```
% a.out > /dev/null ――――――――――― どこにも出力されない
% _
```

このように，coutは単純なディスプレイへの表示を行うわけではなく，標準出力へ出力するものである．

coutを使って作られたプログラムは，同じプログラムを再コンパイルすることなく任意のファイルに出力することができる．

3 標準エラー出力cerr

標準出力の代わりに**標準エラー出力**に出力することもできる．それには，coutの代わりに**cerr**を使う．

注意　標準エラー出力についての詳細は，UNIXの教科書（特にシェル関係）を参照されたい．

【サンプルプログラム】

これは，標準エラー出力への出力をテストするプログラムである．

```
// cerr.C -- 標準エラー出力

#include <iostream.h>

int main()
{
     cerr << "標準エラー出力へ" << endl;
     cout << "標準出力へ" << endl;

    return 0;
}
```

【実行例】

```
% a.out ――――――――――――――――― 普通に実行
標準エラー出力へ                    どちらの出力も端末に表示される
標準出力へ
% a.out > foo ―――――――――――――― 標準出力をリダイレクト
標準エラー出力へ                    標準エラー出力だけは表示される
% cat foo
標準出力へ
% a.out 2> foo ――――――――――――― 今度は標準エラー出力をリダイレクト
標準出力へ
% cat foo
標準エラー出力へ
% a.out >& foo ――――――――――――― 両方をリダイレクト
% cat foo
標準エラー出力へ
標準出力へ
% _
```

4 デバッグへの応用

この節では，coutやcerrをデバッグに応用するときのヒントを説明する．また，次の節では，関連事項として条件コンパイルについて説明する．

4.1◆cout天国

coutでメッセージを表示できるということは，「つまらないこと」あるいは「それほど大したことではないこと」のように見えるかもしれない．しかし，昔の汎用機時代から見ると，自由に端末が使え，自由にメッセージが表示できることは驚異的なことである．

汎用機はCPUタイムが高価だったので，自然とバッチ処理が主流であった．端末も高価だったので，そう自由には使えなかった．

> **注意**
> 1. 「CPUタイムが高価」という意味は，汎用機では使用時間に応じて課金され，しかもその時間当たりの単価がべらぼうに高価であったことを意味する．
> 2. 「バッチ処理が主流」という意味は，1との関係から来る．すなわち，CPUの時間単価が高価なので，対話式に暢気（のんき）に使っているわけに行かず，バッチ処理による自動実行が主流であったという意味である．

汎用機への指示には，パンチされたカードなどが使われた．作成したプログラムがうまく動くかどうかは，プログラムが終了し，結果をプリントアウトするまでわからなかった．

プログラムの実行を開始し，テープが回転すると，ほっとしたものである．とりあえず動いていることがわかったので…．しかし，それ以上のことは，本当にプログラムの実行が終わるまでわからなかったのである．

いまC++が使え，自由に端末が使え，自由にメッセージが表示できるのは天国である．これを利用しない手はない．プログラムの要所要所にcoutを配置し，実行経過を表示するようにするのである．こうすることにより，プログラムが正しく動いているかどうかをチェックすることができる．

4.2◆デバッグへの応用

たとえば，次は第4章で作成した元利合計を求めるプログラムである．

```
#include <iostream.h>

int main()
{
```

```
        cout << 1000 * 1.05 << endl;
        return 0;
}
```

このプログラムは，1000 * 1.0を計算するだけなので，実行してもすぐに終了する．しかし，プログラムによっては，もっと時間のかかるものもある．そこで，次のように関数の入口と出口で関数の名前を表示するようにする．こうすれば，実行時に，いまどこらへんを実行しているのかがわかる．

```
int main()
{
        cout << "main start ..." << endl;  ←―― ここと
        cout << "元利合計= " << 1000 * 1.05 << endl;
        cout << "main end ..." << endl;  ←―― ここ

        return 0;
}
```

4.3◆coutよりもcerr

デバッグ用のメッセージは，coutよりもcerrを使って標準エラー出力に出力したほうがよい．

```
int main()
{
        cerr << "main start ..." << endl;  ←―― cerrに変更
        cout << "元利合計= " << 1000 * 1.05 << endl;
        cerr << "main end ..." << endl;  ←―― cerrに変更

        return 0;
}
```

こうすれば，任意のファイルにリダイレクトしても，デバッグ用のメッセージは端末に表示される．

図5-6▼

```
% a.out
main start ...              ──── mainが開始した
元利合計= 1050              ──── 何か時間のかかる処理をしている
main end ...                ──── mainが終了した
% a.out > foo               ──── 今度はfooにリダイレクト
main start ...              ──── デバッグ用のメッセージだけが
main end ...                     表示される
% cat foo
元利合計= 1050              ──── 本物の処理結果はこちらに
% _
```

5 条件コンパイル

デバッグ用のメッセージは，プログラムが完成したら削除しなければならない．ところが，プログラムのあちこちに散乱するcerrを削除するのは手間がかかる．消し忘れることもある．また，バグが出たとき，再びこれらのメッセージが必要になるかもしれない．

デバッグ時にだけメッセージを表示し，プログラムが完成したらそれを表示しないようにすることができる．しかも，ソースファイルを書き換えることなく….それには，条件コンパイルを使用する．

5.1◆ifdef

条件コンパイルというのは，ある条件を満たしたときだけ，その部分のコンパイルを行うものである．以下に書式を示す．

```
#ifdef NAME
    ...............
    ...............
#endif
```

このように，条件を付ける部分を#ifdefと#endifで囲む．#ifdefの後にあるNAMEは，任意の**記号定数**である．記号定数を定義する方法については，すぐ後

で説明する．

この場合，NAMEという記号定数が定義されていれば#ifdefと#endifで囲まれた部分がコンパイルされる．しかし，定義されていなければコンパイルされない．その部分のプログラムコードは，なかったものとして扱われる．

5.2◆ifndef

条件をひっくり返した，以下のような書き方もできる．

```
#ifndef NAMA
    ...............
    ...............
#endif
```

ifとdefの間にnが追加されたことに注意．この場合は，記号定数NAMEが定義されていない場合だけコンパイルされる．デバッグ用のメッセージは，こちらのほうがよいだろう．完成用のプログラムをコンパイルするよりは，デバッグ用のプログラムをコンパイルするほうが多いからである．

```
#ifndef RUN
    cerr << "main start ..." << endl;
#endif
```

たとえば，このようなコードを普通にコンパイルすると，このcerrは実行される．しかし，プログラム完成時には，記号定数RUNを定義してコンパイルする．すると，このcerrは完成用のプログラムには入らない．

5.3◆サンプルプログラム

次に，元利合計のプログラムを条件コンパイルするように書き換えたものを示す．

【サンプルプログラム】

```
// debug.C -- プリプロセッサ指令

#include <iostream.h>

int main()
```

```
{
#ifndef RUN                          ← 条件コンパイル
    cerr << "main start ..." << endl;
#endif

    cout << "元利合計 = " << 1000 * 1.05 << endl;

#ifndef RUN                          ← 条件コンパイル
    cerr << "main end ..." << endl;
#endif

    return 0;
}
```

5.4◆記号定数の定義（コンパイル時）

コンパイルするとき，記号定数を定義することができる．それには，次のように-Dオプションを使う．

```
g++ -D RUN filename
```

↑ この場合はRUNという記号定数が定義される

> **注意**　#defineを使う方法もあるが，その場合はソースコードの書き換えが必要になる．

次に実行例を示す．

【実行例】

```
% g++ go.C                   ← デバッグ時は普通にコンパイル
% a.out
main start ...               ← デバッグ用のメッセージが表示される
元利合計 = 1050
main end ...
% g++ -D RUN go.C            ← 記号定数RUNを定義してコンパイル
% a.out
元利合計 = 1050              ← デバッグ用のメッセージは表示されない
% _
```

第6章 変数

ディスプレイにデータを表示できるようになると，次はキーボードからデータを入力したくなる．いいかえれば標準入力からデータを読みたくなる．キーボードから入力したデータは，どこかに保存しなければならない．そのデータの受け皿が，**変数**である．

この章では，入力の前段として変数を説明する．それを受けて，第7章では標準入力を説明する．

1 16進数変換プログラム

これからhexというプログラムを作成し，それを改良していく．その過程を通して，**変数**および**標準入力からの入力**を説明する．

1.1◆マニピュレータhex

プログラムを作っていると，時々「10進数→16進数」の変換が必要になる．そこで，「10進数→16進数」の変換を行うhexというプログラムを作成する．

このプログラムを作成するためには，整数を16進数で表示する必要がある．この目的にぴったりのマニピュレータがある．それが**hex**である．作成するプログラムの名前もhexなので混乱しないように！

hexはマニピュレータなので，coutやcerrで<<の右側に置くことができる．

```
cout << … << hex << …
             ↑
            ここ
```

hexが使われると，以後，整数は16進数で表示される．次に例を示す．

【サンプルプログラム】

これは1000と2000という特定の数を16進数に変換するプログラムである．

```
// hex.C -- マニピュレータhexのテスト

#include <iostream.h>

int main()
{
     cout << hex ―――――――――――――――――――― 16進数で出力することを指示
          << 1000 << endl
          << 2000 << endl;
    return 0;
}
```

【実行例】

```
% a.out
3e8 ―――――――――――――――――――――――――――― 10進数の1000
7d0 ―――――――――――――――――――――――――――― 10進数の2000
% _
```

1.2◆マニピュレータdec

もとの10進数で出力するようにするには，マニピュレータの**dec**を使う．

【サンプルプログラム】

これはマニピューレータdecの使い方をテストするためのプログラムである．

```
// hex.C -- hex,decのテスト

#include <iostream.h>

int main()
```

```
{
    cout << hex ———————————16進数で出力することを指示
         << 1000 << endl
         << 2000 << endl
         << dec ———————————デフォルトの10進数に戻すことを指示
         << 1000 << endl
         << 2000 << endl;
    return 0;
}
```

このサンプルプログラムでは，最初に16進数で2つの数値を出力している．その後，基数を10進数に戻し，同じ数値を出力している．

次に，このプログラムの実行例を示す．

【実行例】

```
% a.out
3e8 ——————————————16進数で出力
7d0
1000 —————————————10進数で出力
2000
% _
```

注意　ここでは説明しないが，8進数で出力するにはマニピュレータの「**oct**」を使う．

1.3◆16進数変換プログラム（第1版）

「10進数→16進数」変換プログラムを作成する準備が整ったので，さっそくプログラムの作成に入る．

最初は単純に，変換する数値を固定する．すなわち，変換元になる10進数を固定して，それを16進数に変換する．たとえば，1000という10進数を16進数に変換したければ，次のようなプログラムを作成する．

【サンプルプログラム】

これは汎用的な「10進数→16進数」変換プログラムを作成する前段として，変換する数を1000に固定したものである．

```
// hex.C(第1版) -- 10進数→16進数の変換

#include <iostream.h>

int main()
{
    cout << hex << 1000 << endl;
    return 0;
}
```

実行例

```
% a.out
3e8 ―――――――――――――――― 10進数の1000を16進数で表示した値
% _
```

1.4◆16進数変換プログラム（第2版）

10進数を16進数に変換することができた．しかし，出力結果が見やすくない．

3e8 ――――――――― もとの10進数がいくつかわからない

そこで，もとの10進数を表示して，結果がわかりやすくなるように改良する．

【サンプルプログラム】

```
// hex.C(第2版) -- 10進数→16進数の変換

#include <iostream.h>

int main()
{
    cout << 1000 << " = 0x"
         << hex << 1000 << endl;
    return 0;
```

【実行例】

```
% g++ go.C
% a.out
1000 = 0x3e8 ──────────10進数の1000が16進数の3e8であることがわかる
% _
```

表示的には見やすくなった．しかし，まだまだお世辞にも使いやすいプログラムとはいえない．このプログラムは，変換前の10進数が固定されている．他の数を変換しようとすると，そのたびにプログラムを書き換えて，コンパイルし直さなければならない．

これを改良するため，次のステップとして**変数**を導入する．

2 変数

「変数」というと，数学の方程式を思い出すかもしれない．「具体的な数値の代わりにxという名前の箱を使います．xという箱には好きな数字を代入することができます」

——ぎゃあぁぁぁ…！

数学が嫌いだった人には，変数xは嫌な思い出かもしれない．しかし，プログラミングで使う「変数」も数学の「変数」と同じである．具体的な数で計算する代わりに変数を使って演算を行う．その変数には，任意の数を代入することができる．変数を導入することにより，汎用的なプログラムに仕上げることができる．

——と，理屈はそうである．しかし，多くの人がプログラミングで最初に脱落するのが，この変数でもある．

この節では，その問題の「変数」の基本的な使い方を説明する．しかとして，進もう．

2.1◆宣言

C++のプログラムで変数を使用するには，**変数宣言**が必要である．**宣言**というのは，プログラムの中に名前を導入することである．

宣言　　**プログラムに名前を導入し，その名前をどのように解釈するかをコンパイラに告げること**

第1章の第2.1節で，C++は関数や変数などいろいろな場面で名前を使うことを説明した．これらの名前をどのような目的で使用するかコンパイラに教えてやるのが「宣言」である．変数という名前を使うには，「変数宣言」が必要である．

最初に脅(おど)かすようだが，C++（および，そのもととなったC言語）の変数宣言は，(他の言語のそれに比べ）非常に難しい．なぜなら，C++の変数宣言が**再帰構文**という高度な構造を持っているからである．このため，いくらでも複雑な記述が可能で，上級者でも判読が困難な超難解な宣言を書くことができる．

しかし，それはC++における変数宣言の自由度が高いということであり，書きたければいくらでも複雑な宣言が書けるということである．逆に単純に書くこともできる．最初は単純な書き方を覚え，徐々に複雑な（いいかえれば便利な）書き方を覚えていけばよい．

2.2◆変数宣言

最も単純な変数宣言は，次のように行う．

変数宣言（第1版）

```
型　名前;
```

●型

変数はデータを記憶する．すなわち，変数もデータの一種である．第4章第3節で説明したように，データには「型」がある．

変数宣言では，その変数がどの型のデータを記憶するのかを指定する必要がある．それが，変数宣言の最初に書く「型」である．これは，**型名**という記号を使って書く．たとえば整数の型名はintなので，整数データを記憶する変数を宣言するには次のようにする．

```
int 名前；
```

型名については，次の節で説明する．

●名前

型名の後ろに変数の名前を書く．名前の付け方は，第1章第2.1節で説明した関数の名前の付け方と同じである．たとえばtotalという整数型の変数を宣言するには，次のようにする．

```
int total;
```

2.3◆型名

変数宣言で使う基本的な**型名**を以下に示す．

種類	型名	バイト数
文字	char	1
整数	short	2
	int	2または4
	long	4または8
浮動小数点	float	4
	double	8
	long double	12または16

この表を見ると，同じ整数でもいくつかの種類があるのがわかる．違いはバイト数である．**バイト数**というのは，その変数がメモリ上でどのくらいの大きさを占めるのかを示す値である．バイト数が大きいほど，広い範囲の数を扱うことができる．

2バイトの整数 …… -32768～32767
4バイトの整数 …… -2147483648～2147483647
8バイトの整数 …… -9.2e18～9.2e18（概数）

通常は，intを使う．intより小さい整数を使うときは，shortを使う．intより大きい整数を使うときは，longを使う．

C++は，short，int，longのそれぞれがどれくらいの大きさの整数を扱えるのかを明確には規定していない．使用するコンパイラによって異なる．自分の使用しているコンパイラが各変数をどのサイズで扱っているかを調べる方法については第4.2節で説明する．

しかし，どんなC++コンパイラでも

short≦int≦long

であることは保証されている．shortがintよりも大きくなることはないし，longがintよりも小さくなることはない．

同様に，浮動小数点型にも，いくつかの種類がある．本書では，整数はint，浮動小数点はfloatを使うことにする．以下に変数宣言の例を示す．

```
char c;         ―― 文字型の変数cを宣言
int val;        ―― 整数型の変数valを宣言
float rate;     ―― 浮動小数点型の変数rateを宣言
```

2.4◆代入

変数を宣言したら，その変数に具体的な値を記憶させることができる．これを**代入**という．変数に値を代入するには＝という演算子を使って次のようにする．

変数=値；

以下に例を示す．

```
char c;         ―― 文字型の変数cを宣言し，
c = 'A';        ―― 変数cに文字Aを代入する

int val;        ―― 整数型の変数valを宣言し，
val = 1024;     ―― 変数valに整数1024を代入する

float rate;     ―― 浮動小数点型の変数rateを宣言し，
rate = .25      ―― 変数rateに浮動小数点数の0.25を
                   代入する
```

注意　数学では＝は左辺と右辺が等しいことを示すが，C++は代入を示す．

変数に値を代入すると，その値の代わりにその変数を使用することができる．たとえば，変数valの値が1024のとき，次の文は1024を出力する．

```
cout << val << endl;
        ↑
        1024の代わりになる
```

同じ変数に別の値を代入すると，新しく代入した値がその変数の値になる．次に例を示す．

【サンプルプログラム】

これは，変数と代入をテストするためのプログラムである．

```
#include <iostream.h>

int main()
{
    int n;

    n = 1000; ――――――――― 最初の代入
    cout << "n = " << n << endl;
    n = 2000; ――――――――― 2度目の代入
    cout << "n = " << n << endl;

    return 0;
}
```

【実行例】

```
% a.out
n = 1000 ――――――――― 最初の代入による値
n = 2000 ――――――――― 2度目の代入による値
% _
```

2.5◆変数宣言（2）

変数宣言を行った後，変数に初期値を与えるのが普通である．

```
int n; ――――――――― 変数宣言
n = 1024; ――――――――― 変数に初期値を与える
```

この2行を1行で書くことができる．すなわち，変数宣言と同時に初期値を与えることができる．

```
int n = 1024;
```

次に新しい変数宣言の書式を示す．

変数宣言（第2版）

　　型　名前［＝値］；

書式における［］は，［］の中が省略可能，あるいはオプションであることを示す．

3　16進数変換プログラム（第3版）

変数が使えるようになったので，16進数変換プログラムを，変数を使って書き換える．

【サンプルプログラム】

```
// hex.C(第3版) -- 10進数→16進数の変換

#include <iostream.h>

int main()
{
    int num = 1000;  ――――――――――変数を導入

    cout << num << " = 0x"
         << hex << num << endl;
    return 0;
}
```

【実行例】

```
% a.out
1000 = 0x3e8
% _
```

少し進歩した．というのは，1箇所だけ書き換えれば，他の数の変換ができるようになったからである．

int num = 1000;

↑ ここを書き換えれば他の数を16進数に変換できる

前の版では，2箇所を書き換えなければならなかった．

ここと

```
cout << 1000 << " = 0x"
     << hex << 1000 << endl;
```

ここの2箇所を書き換える必要がある

このように，同じ値を持つリテラルが複数箇所で使われる場合は，変数に置き換えることにより，書き換えが1箇所で済むようになる．これも，変数を導入する効果の1つである．

しかしながら，「他の数を変換しようとすると，そのたびにプログラムを書き換えなければならない」という状況は変わっていない．この改良については，次の章まで待つことにする．

4 sizeof

第2.3節で説明したように，各データ型のサイズは，使用するC++コンパイラによって異なる．ここでは，自分が使用しているC++コンパイラの各データ型のサイズを調べる方法を説明する．

4.1◆sizeof演算子

C++は，**sizeof**という特殊な演算子を持っている．演算子というからには，+や-や*や/と同じ仲間である．演算子はある演算を行い，1つの結果を求める．その演算というのが，指定されたデータがメモリ上に占めるサイズを求めることである．

sizeof演算子には，次の2つの使い方がある．

```
sizeof 変数名
sizeof（型名）
```

このように使用して指定したデータのサイズを求める．以下に例を示す．

sizeof num —— 変数numのサイズを求める
sizeof（int）—— データ型intのサイズを求める

4.2◆データ型のサイズを調べる

sizeof演算子を使えば，charやintといった各データ型のサイズを調べることができる．次のサンプルプログラムは，自分が使用しているC++コンパイラの各データ型のサイズを報告する．

【サンプルプログラム】

```
// sizeof.C -- 各データ型のサイズを求める

#include <iostream.h>

int main()
{
    cout << "char              : " << sizeof(char) << endl
         <<endl;

    cout << "short             : " << sizeof(short) << endl;
    cout << "int               : " << sizeof(int) << endl;
    cout << "long              : " << sizeof(long) << endl
         << endl;

    cout << "float             : " << sizeof(float) << endl;
    cout << "double            : " <<sizeof(double) << endl;
    cout << "long double       : " <<sizeof(long double) << endl;

    return 0;
}
```

【実行例】

```
% a.out
char        :  1    ――― 文字型のサイズ

short       :  2    ――― 整数型のサイズ
int         :  4
long        :  4

float       :  4    ――― 浮動小数点型のサイズ
double      :  8
long double : 12
% _
```

第7章 標準入力

この章では，**標準入力**を説明する．デフォルトで，標準入力はキーボードに結び付けられている．標準入力から入力することにより，キーボードからデータを入力することができる．

1 cin

標準入力からデータを入力するには**cin**を使う．

cin >>受け皿；

この構文により，標準入力（cin）から入力したデータを「受け皿」が受け取る．データを受け取るためには，「受け皿」はリテラルではいけない．変数を使う．変数なら受け取ったデータを記憶しておくことができる．

cin >>変数；

——— cin（標準入力）から入力したデータを変数に代入する

以下に例を示す．

cin >> x;

——— cin（標準入力）から入力したデータを変数xに代入する

【サンプルプログラム】

このサンプルプログラムは，キーボードから入力したデータをそのまま画面に表示する．

```
// cin.C -- cinのテスト

#include <iostream.h>

int main()
{
    int val;  ――――――――――――― データの受け皿となる変数

    cin >> val;  ――――――――――― 標準入力からデータを入力
    cout << val << endl;  ――――― それを標準出力に出力

    return 0;
}
```

【実行例】

```
% a.out
408856  ――――――――――――――― 入力待ちになるので何か入力
408856  ――――――――――――――― 入力したデータが表示される
% _
```

この例では，受け皿となる変数にint型を使ったので，入力できるデータは整数である．「int型変数に対して整数以外のデータが入力されるとどうなるか？」については第5節で説明する．

2 演算子の多重定義

この小さな節は脱線である．

cinは，標準出力coutと対比するとわかりやすい．

cin >>受け皿；
cout <<データ;

cinとcoutでは，使用する記号>>や<<が逆向きになっている．これらの記号は，データが流れる方向を示している．

cin >>受け皿；

――cin（標準入力）から入力したデータを「受け皿」へ流す
（データは左から右へ流れる）

cout <<データ;

――「データ」をcout（標準出力）へ流す
（データは右から左へ流れる）

ちなみに，>>や<<の記号は演算子である．これらの演算子の本来の意味は，以下のとおりである．

演算子	名前	意味
<<	左シフト演算子	ビットパターンを左にシフトする
>>	右シフト演算子	ビットパターンを右にシフトする

C++は大部分の演算子を**多重定義**することができる．演算子が本来持つ意味に別の意味を持たせることができる．

ストラウストラップは，この演算子の多重定義を巧みに使って，ストリームI/Oパッケージを作り上げた．coutやcinをデザインするとき，視覚的に対になっている演算子から<<と>>を選択したそうである．

3 入力可能な16進数変換プログラム

データの入力が使えるようになったので，16進数変換プログラムを書き換え，もとの数を自由に設定できるようにする．

3.1◆16進数変換プログラム（第4版）

次は，16進数変換プログラムで変換する10進数を自由に指定できるようにしたものである．

【サンプルプログラム】

```
// hex.C(第4版) -- 10進数→16進数の変換

#include <iostream.h>

int main()
{
    int num; ―――――――――――― 初期値を与えない

    cin >> num; ――――――――――― もとの数を得る
    cout << num << " = 0x" ――――― 変換して出力
         << hex << num << endl;

    return 0;
}
```

【実行例】

```
% a.out
1000 ―――――――――――― 入力待ちになるので変換したい数を入力
1000 = 0x3e8
% a.out ―――――――――― 再び実行
2000 ―――――――――――― 別の数を変換する
2000 = 0x7d0
% _
```

これで自由に変換したい数を指定できるようになった．ようやく実用的なプログラムに成長した．しかし，まだいくつか改良したい点が残っている．

3.2◆16進数変換プログラム（第5版）

「16進数変換プログラム」を実行すると，黙って入力待ちになる．このことを知らないユーザーは，コンピュータが何を待っているかわからない．そこで，10進数の入力を待っているというメッセージを表示するように変更する．**プロンプト**を表示して入力を促すのである．

【サンプルプログラム】

```
// hex.C(第5版) -- 10進数→16進数の変換

#include <iostream.h>

int main()
{
    int num;

    cout << "10進数? ";   ――― この行を追加
    cin >> num;
    cout << num << " = 0x"
         << hex << num << endl;

    return 0;
}
```

【実行例】

```
% a.out
10進数? 1000   ――― プロンプトが表示される
1000 = 0x3e8
% _
```

4 サンプルプログラム

キーボードからデータを入力できるようになった．練習のため，これまで作ってきた「16進数変換プログラム」以外のプログラムについても，キーボードからデータを与えるように変更する．

4.1◆等差級数

最初は，第4章第2.1節で作成したcount1000.Cを改良する．このプログラムは，1～10000までの合計を求めるものであった．これを1～任意の数までの合計を求めることができるように改良する．

等差級数「1＋2＋…＋x（xは任意の整数)」は，次の式で計算することができる．

1＋2＋…＋x（xは任意の整数）
＝（1＋x）＊x／2

この式を利用すれば，次のようにプログラムすることができる．

【サンプルプログラム】

```
// sum.C --「1 + 2 + ... + x」を計算する

#include <iostream.h>

int main()
{
    int x;

    cout << "「1 + 2 + ... + x」を計算します" << endl
         << "いくつまでの合計を求めますか？";
    cin >> x;
    cout << "1 + ... + " << x << " = "
         << (1 + x) * x / 2 << endl;

    return 0;
}
```

【実行例】

```
% a.out
「1 + 2 + ... + x」を計算します
いくつまでの合計を求めますか？100
1 + ... + 100 = 5050
% _
```

4.2◆オリンピックの開催（第2版）

次は，第4章第2.3節で作成したolimpic.Cを改良する．

オリンピックは4年に1回，4で割り切れる年に行われる．西暦を4で割って余りが0になれば，その年にはオリンピックが開催される．ここでは，任意の西暦を入

力して，その年にオリピックが開催されるかどうか調べることができるように改良する．

【サンプルプログラム】

```
// olimpic.C(第2版) -- オリンピックの開催を調べる

#include <iostream.h>

int main()
{
    int year;

    cout << "西暦? ";
    cin >> year;
    cout << "次の行に表示する数が0ならオリンピックが開催されます"
         << endl
         << year % 4 << endl;

    return 0;
}
```

【実行例】

```
% a.out
西暦? 1998
次の行に表示する数が0ならオリンピックが開催されます
2 ―――――――――― これは2年後にオリンピックが開催されることを意味する
% a.out
西暦? 2000
次の行に表示する数が0ならオリンピックが開催されます
0 ―――――――――― 西暦2000年にはオリンピックが開催される
% _
```

このプログラムは，まだ完成度が低い．なぜなら，出力された数を見て，オリンピックが開催されるかどうかをユーザーに判定させているからである．これは，ちと辛い．きちんとyesかnoで答えるべきである．そのためには，第8章以下で説明する制御文を習得する必要がある．

5 cinの落とし穴

cinを使って入力ができるようになった．しかし，cinには大きな落とし穴がある．それを次のサンプルプログラムで確かめる．

【サンプルプログラム】

これはcinにまつわる落とし穴を調べるためのプログラムである．内容は簡単で，2つの数を入力し，その和を計算するだけのものである．しかし，これがどんなに難しいプログラムであるかが，後で明らかになるだろう．

```
// add.C -- 2数の加算

#include <iostream.h>

int main()
{
    int num1, num2;  ―――――――――― 新しい変数宣言

    cout << "10進数(被演算数) ? ";
    cin >> num1;
    cout << "10進数(演算数) ? ";
    cin >> num2;

    cout << num1 << " + " << num2 << " = " << num1 + num2 << endl;

    return 0;
}
```

以下，これを使って，なぜcinには落とし穴があるのかを説明していく．

5.1◆変数宣言（第3版）

このプログラムは，2つの数を入力し，その和を計算するという簡単なものである．文法的には，新出事項がある．それは変数宣言のしかたで，

```
int num1;
int num2;
```

という2つの変数宣言を次の1本にまとめている．

```
int num1, num2;
```

このように同じ型の変数であれば，複数の変数をカンマで区切ってまとめて宣言することができる．新しい変数宣言の書式を次に示す．

変数宣言(第3版)

型　名前［=値］［,名前［=値］，…］；

この宣言を除けば，あとは単純なプログラムである．2つの数が必要なので，num1，num2を用意し，それぞれ整数を入力した後，加算の結果を表示している．

次に実行例を示す．

【実行例】

```
% a.out
10進数(被演算数) ? 1200 ―――――― num1を入力
10進数(演算数) ? 400 ―――――― num2を入力
1200 + 400 = 1600 ―――――― num1 + num2を出力
% _
```

ここまでは，特に問題ないように見える．

5.2◆cinの落とし穴

問題は，気まぐれなユーザーがいると発生する．まずは，簡単な例から出発する．

【実行例（その1）】

```
% a.out
10進数(被演算数) ? 10
10進数(演算数) ? -7
10 + -7 = 3
% _
```

これは，演算数に**マイナス**を指定した場合である．これ自体は，ユーザーの気まぐれではない．マイナスを入力していけないわけではないのだから．

ただ，結果の表示がうまくない．「10 + -7」のように，プラスの後にマイナスが続いている．これは，いただけない．カッコを付けて「10 + (-7)」のように表示

すべきである．この修正は簡単で，選択文を習えばすぐにでも修正できる．

もっと悲惨なのは，次の例である．

【実行例（その2）】

```
% a.out
10進数(被演算数) ? Hello ―――― 数ではなく文字列を入力
10進数(演算数) ? 0 + 0 = 0 ―――― 何も入力しないのに表示される
% _
```

この例では，ユーザーの気まぐれがいかんなく発揮されている．10進数を入力すべきなのに文字列を入力している．実行例を見ると，その後の演算数の入力をスキップして，いきなり妙な結果が表示されている．

この例の場合はこれで済んでいるが，処理系によっては無限ループになってしまうこともある．

このように，cinでint型データを入力するときは注意が必要である．このエラーを取り除くのは，かなり難しい．反復文の習得まで待つことにする．

注意　cinを何気なく使うのは危険である!!

第8章 選択文if

これまで出てきたプログラムは，すべて**連接**という構造で構成されていた．連接というのは最も単純なプログラム構造で，文は置かれた順に実行される．たとえば，

文1
文2
文3

のようなプログラムでは，「**文1→文2→文3**」の順に実行される．

この章から，もう少し複雑なプログラム構造に挑戦する．新しい構造を習得することによって，どんな複雑な構造を持ったプログラムでも記述できるようになる．

1 基本三構造

オランダの**ダイクストラ**（Edsger W.Dijkstra）は，すべてのプログラムを以下の3つの構造で記述できることを証明した．

連接（*concatenation*）
選択（*selection*）
反復（*repetition*）

これをプログラムの**基本三構造**という．

1.1◆連接

連接は，最も単純なプログラム構造で，文は置かれた順に実行される．

プログラム

```
文1
文2
文3
```

プログラムの実行

文1→文2→文3

1.2◆選択

選択は「場合分け」である．条件により，実行される文が変わる．

プログラム

```
if （条件）
    文1
else
    文2
```

プログラムの実行

（条件）が真なら文1が実行される

（条件）が偽なら文2が実行される

1.3◆反復

反復は，特定の文を繰り返し実行する．

プログラム

```
while （条件）
    文
```

プログラムの実行
（条件）が真の間，文を繰り返し実行する

1.4◆制御文

C++で使われる文は，構造上，次の2種類に分かれる．

単純文——構造を持たない単純な文（連接構造）
制御文——基本三構造のうち「選択構造」と「反復構造」を実現するための文

制御文には，以下の2種類がある．

制御文
選択文
　選択構造を構築するための文
　条件により，実行する文を場合分けする
反復文
　反復構造を構築するための文
　条件により，同じ文を繰り返し実行する

制御文以外の文が**単純文**である．

この章と次の章では，制御文のうち，選択文を説明する．選択文を習得することにより，基本三構造のうちの選択構造を記述できるようになる．反復文については，その後の章で説明する．

2 場合分け——if

第7章第4.2節で作成したolimpic.Cは，西暦を入力するとその年にオリンピックが開催されるかどうかを判定してくれるというものであった．ただし，プログラムの出力は0～3のいずれかの数字が表示されるだけで，ユーザーがその数字を見て開催の有無を判定しなければならなかった．

これをもう少しわかりやすいプログラムにしようとすると，次のような場合分けの処理が必要になる．

結果が0ならば次の文を実行する

```
cout << "開催されます" << endl;
```

そうでなければ次の文を実行する

```
cout << "開催されません" << endl;
```

つまり，条件によって実行する文を場合分けする必要がある．単純文では，このような芸当はできない．**if文**という**選択文**を使うと，このような場合分けが可能になる．

2.1◆if文

if文には，2つのバリエーションがある．以下に，最も単純な形を示す．

if文の書式

```
if (式) ――――― 式の値が真ならば
   文 ――――― この文が実行される
```

ifの後ろに（）を置き，（）の中に直後の文を実行するかどうかを決めるための式を書く．式の値が「真」なら直後に置かれた文が実行される．しかし，式の値が「偽」の場合は，直後に置かれた文が実行されない．

2.2◆式の真偽

式というのは「値を持つもの」といった意味である．C++の式は，以下のように真偽が判定される．

真偽の判定

式の値が0 ――――― その式は偽
式の値が0ではない ――――― その式は真

次に，いくつか例を示す．

```
if (3 + 2) ――――― 3 + 2は0ではないので真
   文 ――――― したがって，この文は実行される
```

```
if (3 + 2 - 5) ———— 3 + 2 - 5は0なので偽
    文 ———— したがって，この文は実行されない

if (-100) ———— 100は0ではないので真
    文 ———— したがって，この文は実行される

if (0) ———— ( )の中は0なので偽
    文 ———— したがって，この文は実行されない
```

2.3◆サンプルプログラム

次のサンプルプログラムは数を入力させ，その値をif文の真偽の判定に使っている．0以外の数を入力すると真と判定され，その旨を示すメッセージが表示される．しかし，0を入力すると何も表示されない．

【サンプルプログラム】

```
// true.C -- 入力した値の真偽を調べる

#include <iostream.h>

int main()
{
    int num;

    cout << "数? ";
    cin >> num;

    if (num) ———— 入力した値を真偽の判定に使う
        cout << num << "は真です" << endl;

    return 0;
}
```

【実行例】

```
% a.out
数? 1234 ———— 1234は0でないので真
1234は真です
% a.out
数? 0 ———— 今度は0（偽）を入力
% _ ———— 何も表示されない
```

2.4◆else節

if文に**else節**を追加することができる．else節を付けると，条件が「偽」の場合に実行する文を書くことができる．これがif文の2つ目のバリエーションである．

if文の書式（その2）

if (式)	式の値が真ならば
文1	この文が実行される
else	式の値が偽ならば
文2	この文が実行される

この書式を使うと，式の真偽によって**文1**か**文2**のどちらか一方を実行させることができる．両方の文が同時に実行されるということはない．

この書式の応用として，olimpic.Cの改訂版を，次の第3節で作成する．

2.5◆if文のインデント

if文は，全体で1つの文である．したがって，1行に収めて書くのが原則である．

```
if (式) 文;
```

たとえば，次のように．

```
if (num) cout << num << "は真です" << endl;
```

しかし，これでは読みにくい．そこで，条件が真の場合に実行される文を2行目に書くのが普通である．しかも，その文を右に**インデント**（段付け）して書く．

```
if (num)
    cout << num << "は真です" << endl;
```

何文字分インデントするかは自由である．本書では，4文字ずつインデントする．

if～elseも1行に書くとわかりにくい．

```
if (式) 文1 else 文2
```

そこで，次のように3行に分けて書くとわかりやすい．

```
if (式)
    文1
else
    文2
```

本書では，上のような書き方をする．

3 オリンピックの開催（第3版）

第7章第4.2節で作成したolimpic.Cにif～else文を導入し，出力結果がわかりやすくなるように改良する．

西暦を4で割った余りが0なら，その年にオリンピックが開催される．一方，if文の条件式は0かどうかで「真偽」が判定される．そこで，西暦を4で割った余りを求める式をif文の（）の中に置く．こうすれば，オリンピックが開催されるかどうかで場合分けをすることができる．

```
if (year % 4)
    文1 ―――― 非0の場合に実行される
                （%は剰余を求める演算子）
else
    文2 ―――― 0の場合に実行される
```

この場合，オリンピックが開催されると「year % 4」が0になるので，偽と判定されて**文2**が実行される．つまり，次のような感じになる．

```
if (year % 4)
    cout << "オリンピックは開催されません";
else
    cout << "オリンピックは開催されます";
```

【サンプルプログラム】

これは，olimpic.Cにif～else文を組み込んでわかりやすい表示に修正したものである．

```
// olimpic.C(第3版) -- オリンピックの開催を調べる

#include <iostream.h>

int main()
{
    int year;

    cout << "西暦? ";
    cin >> year;

    if (year % 4)
        cout << "オリンピックは開催されません" << endl;
    else
        cout << "オリンピックは開催されます" << endl;

    return 0;
}
```

【実行例】

```
% a.out
西暦? 1998
オリンピックは開催されません
% a.out
西暦? 2000
オリンピックは開催されます
% _
```

これまでのolimpic.Cに比べ，ずいぶんわかりやすいプログラムに改良されている．

4 条件式

if文の（）の中には真偽を判定するための式を書く．これを**条件式**という．また，後から出てくるいろいろな制御文も条件式を使用する．この節では，いろいろな条件式の書き方をまとめる．

4.1◆演算子とオペランド

加算で使う + のように演算の種類を決める記号を**演算子**（*operator*）という．また，演算の対象となる値を**オペランド**（*operand*）という．

1000 + 2000

たとえば，この例では + が演算子，1000と2000がオペランドである．

演算の種類によってオペランドの数が異なる．C++で使う演算は，オペランドの数により，以下の3種類に分かれる．

単項演算
【例】-100

二項演算
【例】100 - 50

三項演算
【例】(x > y) ? x : y

三項演算については，第9章で説明する．

4.2◆算術式

演算結果が数になる演算を**算術演算**という．具体的には，第4章第2節で説明した**四則演算**が算術演算である．

演算子	演算
+	加算
-	減算
*	乗算
/	除算
%	剰余

算術演算における式の真偽は，すでに説明した通り「0」か「0でない」かで決まる．

真偽の判定

式の値が0 ―――― その式は偽
式の値が0ではない ―――― その式は真

4.3◆論理否定演算子！

論理否定演算子！を使って式の「真偽」を逆転することができる．たとえば，

```
if (式)
    文 ―――― 式が真の場合に実行される
```

なら，式の値が「真」のとき文が実行される．しかし，これを次のように書けば，式の値が「偽」のときに文を実行させることができる．

```
if (!式)
    文 ―――― 式が偽の場合に実行される
```

たとえば，除算を行うプログラムがあるとする．除算を行うときに注意しなければならないのは，「0で割ることはできない」ということである．

そこで，入力した数をチェックし，除数に0が入力されたらエラーメッセージを表示するようにする．除数がnum2とすると，次のような感じになる．

```
if (num2)
    cerr << "0で割ることはできません!" << endl;
```

ただし，これでは真偽が逆である．num2が0のとき，このメッセージを表示させたいのであるが，0は偽なのでcerrは実行されない．そこで，！を使って真偽を逆転する．

```
if (!num2)
    cerr << "0で割ることはできません!" << endl;
```

else節を持つif文でも！を使って真偽を逆転できる．すなわち，普通は

```
if (式)
    文1 ―――― 式が非ゼロのとき実行される
else
    文2 ―――― 式がゼロのとき実行される
```

のように書くところを，！を使うと次のようになる．

```
if (!式)
    文1 ―――――――――― 式がゼロのとき実行される
else
    文2 ―――――――――― 式が非ゼロのとき実行される
```

5 結果が「真/偽」になる演算

if文の条件式に3 + 4などの算術式をそのまま置くことは少ない．むしろ，次のような演算を書くことが多い．

等値演算 ―――――――――― 左辺と右辺が等しいか比較する
関係演算 ―――――――――― 左辺と右辺の大小を比較する
論理演算 ―――――――――― 「かつ」，「または」など

これらの演算に共通しているのは，演算結果が「真偽」のいずれかになるということである．したがって，if文の条件式としては，これらの結果が「真/偽」になる演算を置くほうが自然で読みやすくなる．

5.1◆等値演算

まず**等値演算**から始める．

演算子	演算
x == y	xとyが等しければ真
x != y	xとyが等しくなければ真

数学では=が等値を表すが，C++では，すでに=を代入演算に使用している．そこで，代わりに==を等値演算子に使用する．以下に例を示す．

```
if (num == 100) ――― 変数numの値が100に等しければ真
if (num != 100) ――― 変数numの値が100に等しくなければ真
```

この2行目は，論理否定演算子の！を使って，次のように書くこともできる（あまり使わないが…）．

if (!(num == 100))

==も!=も「真」か「偽」のいずれかの値を返す．実際，次のサンプルプログラムを実行すると，それを確かめることができる．

【サンプルプログラム】

これは，==演算子の演算結果を表示するプログラムである．

```
// eq.C -- == の値を表示する

#include <iostream.h>

int main()
{
    cout << (100 == 100) << endl ───── 「真」がいくつになるか調べる
         << (100 == 200) << endl; ──── 「偽」がいくつになるか調べる

    return 0;
}
```

これは，(100 == 100) や (100 == 200) の値を直接coutで表示している．(100 == 100) は等しいので「真」に相当する値を表示するはずである．また，(100 == 200) は等しくないので「偽」に相当する値を表示するはずである．

このプログラムを実行すれば，これらの値がいくつになるのか調べることができる．次に，その実行例を示す．

【実行例】

```
% a.out
1 ◀──────── (100 == 100) の値
0 ◀──────── (100 == 200) の値
% _
```

この結果を見ると，==演算子は「真」の値を1（1は非ゼロである）として扱っているのがわかる．また，「偽」の値を0として扱っているのがわかる．!=も同様に，真偽によって1か0の値を返すことを確かめることができる．

5.2◆関係演算子

関係演算子を使うと，大小の比較によって「真偽」を判定できる．

演算子	演算
x < y	xがyよりも小なら真 そうでなければ偽
x <= y	xがy以下なら真 そうでなければ偽
x >= y	xがy以上なら真 そうでなければ偽
x > y	xがyよりも大なら真 そうでなければ偽

次に例を示す.

【サンプルプログラム】

このプログラムは，2つの整数を入力し，大きい方の整数を表示する.

```
// max.C --2つの整数の大小を比較する

#include <iostream.h>

int main()
{
    int num1, num2;

    cout << "2つの数の大小比較をします" << endl
         << "1つ目の数? ";
    cin >> num1;
    cout << "2つ目の数? ";
    cin >> num2;

    if (num1 < num2)
        num1 = num2;
    cout << num1 << "のほうが大きいです" << endl;

    return 0;
}
```

【実行例】

```
% a.out
2つの数の大小比較をします
1つ目の数? 3
2つ目の数? 2
3のほうが大きいです
% _
```

このプログラムのポイントは，num1のほうが大きいと仮定していることである．ただし，それでは正しい答えにならない．そこで，if文を使ってnum1とnum2を比較し，num2のほうが大きい場合はnum1にnum2を代入している．これで，いつでもnum1のほうが大きいことになる．

5.3◆論理演算子

論理演算子は，複数の条件式を「かつ」「または」で結合したいときに使う．

演算子	演算
x && y	論理積（かつ） 式xと式yがともに真なら真 そうでなければ偽
x || y	論理和（または） 式xと式yのいずれか一方が真なら真 そうでなければ偽

たとえば，整数変数mに代入されている値がサイコロの目としてふさわしいかをチェックしたいとする．サイコロの目は1～6のいずれかの整数なので，次の2つのチェックが必要である．

```
if (1 <= m)
if (m <= 6)
```

論理演算子&&を使えば，これら2つの条件を結合することができる．

```
(1 <= m) && (m <= 6)
```

この式は，「mが1以上」かつ「mが6以下」なら真という意味になる．したがって，次のようなif文を作ることができる．

```
if ((1 <= m) && (m <= 6))
    cout << "mはサイコロの目です" << endl;
```

論理演算子||を使えば，逆の表現もできる．

```
(m < 1) || (6 < m)
```

この式は，「mが1より小」または「mが6より大」なら真という意味になる．したがって，次のようなif文を作ることができる．

```
if (m < 1) || (6 < m)
    cout << "mはサイコロの目ではありません" << endl;
```

&&と||では，ちょうど条件の付け方が逆になることに注意．なお，論理演算の面白い使い方を第9章の第5節で紹介するので参照されたい．

6 ブロック

わかっているつもりでも，いざ実践してみると思わぬ壁にぶち当たってしまうのがプログラミングである．

たとえば，除算を行うプログラム．除算なんてちょろい，ちょろい．やり方は全部わかっている．除算の演算子は/だし，除数がゼロの場合のチェックのしかたもわかっている．

で，実際にプログラムを書いてみる．しかし，ここでハタと困ってしまう．いままでの知識では不可能であるこことが判明する．

この節では，乗算プログラムを例に**ブロック**を説明する．

6.1◆プログラムの落とし穴

それでは，実際に除算を行うプログラムを作ってみる．まず，除算を行うのだから，オペランドが2つ必要である．そこで，変数を2つ用意する．

```
int num1, num2;
```

そして，cinで整数を入力させる．

```
cin >> num1;
cin >> num2;
```

と，ここまではよい．num2はゼロではまずいから，前の節で習ったようにチェックを入れる．

```
if (!num2)
    cerr << "0で割ることはできません!" << endl;
```

ここで，困ったことが起こる．num2のチェックはできた．しかし，もしnum2が本当にゼロなら，この先の計算に進まれては困る．つまり，次のようにnum2がゼロならエラーメッセージを表示して，プログラムを終了させたい．

```
if (!num2)
    cerr << "0で割ることはできません!" << endl;
    //ここでプログラムを終了させる
```

ところが，if文では，条件による文を1つしか実行できない．

```
if (式)
    文 ―――――――― ここには文を1つしか置けない
```

条件が「真」の場合に2つ以上の文を実行させたくても，それはできない．また，仮にそれができたとして，プログラムの途中でプログラムを終了させるにはどうすればよいのか？

つまり，たかが除算プログラムでも，まだまだ知らないことがあるのである．プログラムを甘く見てはいけない．

6.2◆プログラムの終了

それでは，まずプログラムの途中でプログラムを終了させる方法から見ていこう．それには，C++におけるプログラムの実行の流れを見ればよい．

プログラムの実行開始位置

main関数の最初の文

プログラムの終了

main関数の最後の文を実行
exit関数を実行
main関数の中でreturn文を実行

プログラムを終了させる方法はいろいろあるが，代表的なのは上に挙げた3つである．

一般にプログラムの途中で**exit**という関数を使うが，まだわれわれの知識では，この関数を使うことはできない．そこで，ここではreturn文を使う．

return文というと，すでにこれまで何度も使用してきた．main関数の中でreturn文を使うと，exit関数を呼び出す効果がある．したがって，main関数の途中にreturn文を置けば，そこでプログラムを終了させることができる．

```
int main()
{
    文1
    文2
    return;———————ここでプログラムが終了する
    文3
    文4
}
```

たとえば上の場合，**文1→文2**と実行したところでプログラムが終了する．**文3**と**文4**は実行されない．

プログラムの途中で終了させたいときは，このようにreturn文を使えばよい．ただし，この方法は関数mainの中でしか使えない．

6.3◆ブロック

次に，if文に複数の文を置く方法を考える．これは，**ブロック**を使えば解決する．ブロックというのは，次のような形をしている．

```
{ }
```

要するに，一組の中カッコ（ブレース）である．これだけでは何の役にも立たないが，ブロックの中には複数の文を置くことができる．

```
{文1文2文3…}
```

ただし，これでは読みにくいので，普通は次のように行を分けて書く．

```
{
    文1
```

```
    文2
    文3
    …
}
```

main関数で使う {} もブロックである．

ブロックの中に置かれた文は，先頭から順に実行される．たとえば次のようなプログラムがあったとする．

```
文1
文2
{ ←――――――――― ここから
        文3
        文4
} ←――――――――― ここまでがブロック
文5
```

この場合，プログラムは「**文1→ 文2→ 文3→ 文4→ 文5**」のように実行される．つまり，上のプログラムは次のプログラムと同じである．

```
文1
文2
文3
文4
文5
```

「だったら {} を付けても付けなくても同じじゃないか．{} を入力する分，無駄である」――ごもっともである．しかし，ブロックが重要なのは，ブロック全体で1つの文として扱われることである．ブロックを使えば，1つの文しか置けないところに複数の文を置くことができる．

```
if (式)
    文 ――――――――― ここにブロック文を置く
```

すると，次のようになる．

```
if (式)
    {文1文2文3…}
```

これを普通は次のように書く．

```
if (式) {
    文1 ―――――――― 式が真のときブロックの中の文が実行される
    文2
    文3
    …
}
```

これで，if文に複数の文を置くことができるようになる.

6.4◆除算プログラム

それでは，除算プログラムを完成させよう．次に完成版のプログラムを示す．変数をintで宣言したので，対象は整数である．せっかく整数どうしの除算を行うのだから，剰余（余り）も求めるようにした.

【サンプルプログラム】

```
// div.C -- 除数がゼロなら終了する

#include <iostream.h>

int main()
{
    int num1, num2;

    cout << "整数の除算を行います" << endl
         << "1つ目の整数? ";
    cin >> num1;
    cout << "2つ目の整数? ";
    cin >> num2;

    if (!num2) {
        cerr << "0で割ることはできません!" << endl;
        return 1;
    }

    cout << num1 << " ÷ " << num2 << " = "
         << num1 / num2 << "…"
         << num1 % num2 << endl;

    return 0;
}
```

【実行例】

```
% a.out ――――――――――――――――――― 第1回目の実行
整数の除算を行います
1つ目の整数? 13
2つ目の整数? 5
13 ÷ 5 = 2…3
% a.out ――――――――――――――――――― 第2回目の実行
除算を行います
1つ目の数? 10
2つ目の数? 0 ―――――――――――――― ゼロを入力すると，はじかれる
0で割ることはできません!
% _
```

7 代入式

C++の式で間違えやすいのは，= と == である．前者は「代入」であり，後者は「等値」である．ところが，数学では = を「等しい」という意味で使用する．この長い間の習慣から，うっかり次のような条件式を書いてしまうことがある．

```
if (num = 100)
```

本人は「numが100に等しければ…」のつもりで書いたものだが，これでは「numに100を代入する」という意味になってしまう．

この節では，= と == に関わる注意事項を扱う．

7.1◆文と式

本人は「等しければ」のつもりでうっかり書いてしまいやすい次のようなif文――

```
if (num == 100) ―――――― 本当はこのように書くつもりが
  ↓
if (num = 100) ―――――― = の数が1つ足りない
```

では，「この書き方は誤りか？」というと，C++の文法からいえば誤りではない．コンパイラを通ってしまう．なぜなら，C++では代入は式なので，普通の式と同じ

ように条件文の（）の中に書くことができる．

代入は「文」ではなく「式」である!!

この違いは重要である．文はそれ自身で完結しているが，式は文の構成要素であり，文の一部である．文は値を持たないが，式は値を持つ．

式なら別の式のオペランド（演算の対象）にすることができる．たとえば，次のような式を書くことができる．

```
(num1 = 100) + (num2 = 200)
```

これは，代入式どうしを加算している．もし，代入が文なら，こんな記述はできない．

7.2◆代入式の値

代入が式であるとすると，値を持つことになる．では，代入式の値は何だろう？

これは，代入後の変数の新しい値である．代入が行われ，その結果，変数の値が新しくなる．その新しい値が代入式の値になる．

```
num = 100
      ─────── numの新しい値は100
              したがって，この式の値は100

(num1 = 100) + (num2 = 200)
      ─────── (mum1 = 100)の値は100
              (num2 = 200)の値は200
              したがって，この+式は100 + 200で300になる
```

これで，if（num = 100）の意味がわかるだろう．この条件式の値はnumの新しい値――すなわち，100である．100は非ゼロなので「真」である．したがって，この条件式は「真」と判定される．

```
if (num = 100)────── 条件式は真
    文 ────────────── したがって，この文はつねに実行される
```

さらにelse節を持つ次のif文は悲惨である．**文2**が実行されることは永久にない（なぜか？）．

```
if (num = 100)
    文1
else
    文2
```

このように条件式で = と == を間違えると，文法的には誤りでないのでコンパイルを通過してしまう．このため，誤りに気づきにくいので，慣れるまでは注意が必要である．

7.3◆誤らないために

もし，どうしても == と = を間違えたくなければ，次のように書く方法もある．

```
if (100 == num)
```

すなわちリテラルを == の左辺に書いてしまうのである．これでも「変数numの値が100に等しければ」の意味になる．このとき == を = に間違えても大丈夫である．

```
if (100 = num)
```

100はリテラルなので，新しい値を代入することはできない．したがって，この文はコンパイルエラーになる．エラーになれば，いやでも = が1つ足りないことに気づくだろう．

7.4◆便利な代入式

代入が式であることを強調するとき**代入式**という．

なぜC++では，代入が式なのだろう？　代入が文であったなら == と = を間違わないで済んだのに….それは，代入が式なら便利だからである．2つほど例を示す．

●プログラムの簡略化

プログラムでは，代入を行い，その値によって場合分けを行うということがよくある．

```
num =…;       ―― 変数numに何かを代入する
if (num)      ―― 変数numの値が真なら
```

代入式を使えば，これを次のように簡略化して書くことができる．

```
if (num =…)
```

●代入の結合

同じ値を複数の変数に代入する場合，1つの式に結合することができる．たとえば，次のような代入を行いたいとする．

```
x = 100;
y = 100;
```

これを，次のようにまとめて書くことができる．

```
x = y = 100;
```

これは，代入式の**結合規則**が「右から左」だからである．結合規則については，項を改めて説明する．

7.5◆結合規則

結合規則というのは，同じ優先順位の演算子を複数個使った場合，左右どちらの演算子から評価を始めるかを示すものである．たとえば次のような式があったとする．

x OP y OP z（OPは，ある演算子）

この式には，2つのOPが結合されている．もしOPの結合規則が「左から右」なら，次のように左側の演算――すなわち，x OP yが先に評価される．

(x OP y) OP z

逆にOPの結合規則が「右から左」なら，次のように右側の演算――すなわち，y OP zが先に評価される．

x OP (y OP z)

結合規則は，演算の種類によって異なる．たとえば，次のように2つの代入演算子が同時に使われたとする．

```
x = y = 100;
```

これを見てびっくりしないように！　代入式の**結合規則**は「右から左」である．したがって，複数の代入式が同時に使われた場合，右側の代入式が先に評価される．つまり，上のような式では，右側のy = 100が先に評価される（実行される）．

```
x = (y = 100);
```

まず、こちらの代入が先に行われる

この結果，yに100が代入され，さらに式の値が100になる．その100がxにも代入されるので，結局xにもyにも100が代入されることになる．

```
x = (y = 100);   ――― まずyに100が代入される
x = (100)        ――― 代入式全体の値が100になる
x = 100          ――― この結果、xにも100が代入される
```

7.6◆式文

繰り返し説明したように，**式**と**文**は異なる．文はそれ自身で完結しているが，式は文の一部であり，それだけでは文として成立しない．

ただし，式を文にする方法がある．それが**式文**である．式を式文にするには，式の後にセミコロンを付ければよい．

```
式   ――― これは文ではない
式;  ――― これは文である（式文）
```

式文は，式の評価それ自体を文として完結させたいときに使用する．その代表的な例が代入である．代入は，それ自体が意味を持つ．しかし，代入だけでは文にならない．そこで，最後にセミコロンを付けて文にする．

```
x = 100   ――― これは式であって完結した文ではない
x = 100;  ――― これなら文になる
```

p.130の「**演算子のトリック**」のところで，次のような文を示す予定である（第9章第5節）．

```
num1 && (num2 = 100);
```

これも式文である．&&だけでは論理演算式にしかならない．しかし，最後にセミコロンを付けることにより，完結した文にすることができる．

第9章 高度な選択文

前の章では，選択文の代表としてif文を説明した．この章では，その他の選択文や，やや高度な選択文を扱う．

1 3つ以上の場合分け(1)―― else if

if文では，たかだか2通りの場合分けができる．しかし，ときには3つ以上の場合分けが必要になることがある．この節と次の節で，3つ以上の場合分けについて説明する．

1.1◆else if

if文では，たかだか2通りの場合分けができる．

```
if (式)
    文1    // 場合1
else
    文2    // 場合2
```

else節に置く**文2**は，文なら何でもよい．そこで，**文2**に再びif文を使えば，3通りの場合分けが可能になる．

```
if (式)
    文1
```

```
else
    文2 ―――――――― この文2をif文にする
 ↓
if (式)
    文1
else
    if (式)
        文2
    else
        文3
```

これをもう少しわかりやすく書き直すと，以下のようになる（C++は自由書式が許されていることを思い出すこと）．

```
if (式)
    文1    // 場合1
else if (式)
    文2    // 場合2
else
    文3    // 場合3
```

このようにelse節にif文をネストさせることにより，何通りもの場合分けが可能になる．

```
if (式)
    文1    // 場合1
else if (式)
    文2    // 場合2
else if (式)
    文3    // 場合3
else if (式)
    文4    // 場合4
……
```

1.2◆遊園地の入園料

次に，4通りの場合分けが必要な例を示す．以下のサンプルプログラムは，ある

遊園地の入園料を表示するものである．この遊園地では，年齢により入園料が4段階に分かれている．

区分	年齢	料金
乳児	4歳以下	無料
幼児	7歳以下	400円
小学生	12歳以下	600円
大人	13歳以上	1000円

【サンプルプログラム】

これは，4段階に分かれた遊園地の入園料を求めるプログラムである．else ifのサンプルプログラムになっている．

```
// admission.C -- 入園料の計算

#include <iostream.h>

int main()
{
    int old;

    cout << "年齢? ";
    cin >> old;

    if (old <= 4)
        cout << "無料です" << endl;      // 場合分け1
    else if (old <= 7)
        cout << "400円です" << endl;     // 場合分け2
    else if (old <= 12)
        cout << "600円です" << endl;     // 場合分け3
    else
        cout << "1,000円です" << endl;  // 場合分け4

    return 0;
}
```

【実行例】

```
% a.out
年齢? 4 ―――――――――――――― 4歳以下の入力例
無料です
% a.out
年齢? 7 ―――――――――――――― 7歳以下の入力例
400円です
% a.out
年齢? 12 ――――――――――――― 12歳以下の入力例
600円です
% a.out
年齢? 13 ――――――――――――― 13歳以上の入力例
1,000円です
% _
```

2 3つ以上の場合分け(2)── switch

場合分けをするとき，if文の代わりに**switch文**を使うこともできる．特に3つ以上の場合分けを行うときは，if文よりもswitch文のほうがきれいに書ける．

ただし，switch文では条件の書き方に制限がある．ある値と定数（定数式）との比較しかできない．その点では，if文の条件式のほうが自由度が高い．

2.1◆switch文

switch文は，switch，case，defaultの3つのキーワードを使って，次のように書く．

【switch文の書式】

```
switch (定数式) {
case定数1:
    文の並び1
    break;
case定数2: ―――――――――――― 定数式の値が定数2に一致したら
    文の並び2 ――――――――――― これらの文が実行され
    break; ―――――――――――― ここでswitch文を抜ける
case定数3:
```

```
        文の並び3
        break;
………
default: ────────────────── 定数式がそのcaseにも当てはまらないときは
        文の並び ────────────────── これらの文が実行される
}
```

（）で囲まれた式の値が，caseに続く定数（結果が定数となる式なら，定数でなくてもよい）と一致すると，その後に続く文をbreak，または｝が現れるまで実行する．たとえば，式の値が**定数2**と一致すれば**文の並び2**が実行される．

default:の部分には，どのcaseにも当てはまらないときに実行する処理を書く．default節はオプションなので，なくてもよい．

2.2◆簡易メニュー

次にswitch文を使ったサンプルプログラムを示す．

【サンプルプログラム】

このプログラムは簡単なメニューの作り方を示すものである．ユーザーが入力した番号で場合分けを行い，それぞれのメニューコマンドを実行する．

```
// menu.C -- 簡易メニュー

#include <iostream.h>

int main()
{
    int select;

    cout << "メニュー" << endl
         << "-----------------" << endl
         << "1]ファイルをロード" << endl
         << "2]ファイルをセーブ" << endl
         << "0]終了" << endl
         << "-----------------" << endl
         << "選択? ";
    cin >> select;

    switch (select) {
    case 1:
```

```
        cout << "ファイルをロードします" << endl;
        break;
    case 2:
        cout << "ファイルをセーブします" << endl;
        break;
    case 3:
        cout << "プログラムを終了します" << endl;
        break;
    default:
        cout << "番号が不正です" << endl;
    }

    return 0;
}
```

【実行例】

```
% a.out ―――――――――――――― 第1回目の実行
メニュー
-----------------
1]ファイルをロード
2]ファイルをセーブ
0]終了
-----------------
選択? 2 ―――――――――――――― 2を選択
ファイルをセーブします
% a.out ―――――――――――――― 第2回目の実行
メニュー
-----------------
1]ファイルをロード
2]ファイルをセーブ
0]終了
-----------------
選択? 3 ―――――――――――――― 今度は3を選択
プログラムを終了します
% a.out ―――――――――――――― 第3回目の実行
メニュー
-----------------
1]ファイルをロード
2]ファイルをセーブ
0]終了
-----------------
選択? 4 ―――――――――――――― このメニューはない
番号が不正です
% _
```

2.3◆バリエーション（1）

switch文でcase節の後ろに文を置かないときがある．

case定数:

←――― ここに文を置かない

その場合は，次のcase節で一致したものと見なされる．

case定数1: ――― ここで一致すると続く文がないので
case定数2: ――― ここで一致したと見なし
　文の並び ――― これらの文が実行される
　break;

つまり，上の場合では**定数1**で一致しても**定数2**で一致しても，**文の並び**が実行される．

このように，case節の後ろの文を省略すると，複数の値に一致する処理を書くことができる．

case定数1:
case定数2:
case定数3:
　文の並び ――― 定数1～定数3のどれに一致しても
これらの文が実行される

次に例を示す．

【サンプルプログラム】

これは，1～6の数について偶数／奇数の判定を行うプログラムである．case節の後ろの文を省略することにより，複数の定数をグループ化している．

```
#include <iostream.h>

int main()
{
    int num;

    cout << "1～6までの数? ";
    cin >> num;
    switch (num) {
```

```
    case 1:  ←――――――――――― 注目
    case 3:
    case 5:
        cout << num << "は奇数です" << endl;
        break;
    case 2:  ←――――――――――― 注目
    case 4:
    case 6:
        cout << num << "は偶数です" << endl;
        break;
    default:
        cout << num << "は不正な数です" << endl;
    }

    return 0;
}
```

【実行例】

```
% a.out ―――――――――――――――― 第1回目の実行
1〜6までの数? 3
3は奇数です
% a.out ―――――――――――――――― 第2回目の実行
1〜6までの数? 6
6は偶数です
% a.out ―――――――――――――――― 第3回目の実行
1〜6までの数? 7
7は不正な数です
% _
```

2.4◆バリエーション (2)

switch文のcase節でbreak文を置かないときがある．

```
case定数:
    文の並び
    ←―――――――――― ここにbreak文を置かない
```

その場合は，次のcase節の文も実行する．

```
case定数1:  ―――― ここで一致すると
    文の並び1  ―――― これらの文と
case定数2:
    文の並び2  ―――― これらの文も実行される
    break;
```

このように，次のcase節の文も実行させたいときは，break文を省略する．

【サンプルプログラム】

これは，case節でbreak文を省略した場合のサンプルプログラムである．case 1の場合，break文がないので，case 2の文も実行される．

```
#include <iostream.h>

int main()
{
    int select;

    cout << "メニュー" << endl
         << "-----------------" << endl
         << "1] 2本足で立つ" << endl
         << "2] 火を使う" << endl
         << "3] 4本足で歩く" << endl
         << "-----------------" << endl
         << "選択? ";
    cin >> select;

    switch (select) {
    case 1:
        cout << "直立歩行する" << endl;  ―――― 後ろにbreak文がない
    case 2:
        cout << "人間です" << endl;
        break;
    case 3:
        cout << "動物です" << endl;
        break;
    default:
        cout << "番号が不正です" << endl;
    }

    return 0;
}
```

【実行例】

```
% a.out
メニュー
-----------------
1] 2本足で立つ
2] 火を使う
3] 4本足で歩く
-----------------
選択? 1
直立歩行する ―――――――――――― case 1の文
人間です ―――――――――――――― case 2の文
% _
```

1を選択すると，case 1とcase 2の両方の文が実行されているのがわかる．

3 if文のネスト

if文をネストすると（多重に使うと），ifとelseの対応がわかりづらくなる．たとえば，次のようにif文を2つ重ねた場合，else節はどちらのif文に対応するのだろう？

```
if (hight >= 170) ―――――――――――――――――― (1)
if (weight > 80) ――――――――――――――――――― (2)
    cout << "あなたはもてません" << endl;
else
    cout << "あなたはもてる可能性があります" << endl;
```

3.1◆ifとelseの対応

もしelse節が（1）のifに対応するなら，上のような意味になる．

```
if (hight >= 170) ―――――――――――――――――― (1)
   if (weight > 80) ――――――――――――――――― (2)
     cout << "あなたはもてません" << endl;
else
     cout << "あなたはもてる可能性があります" << endl;
```

この場合は，（奇妙なことに）体重に関わりなく身長が169cm以下であれば，もてる可能性がある．しかし，(2) のifに対応するなら下のような意味になる．

```
if (hight >= 170) ―――――――――――――――― (1)
    if (weight > 80) ―――――――――――――― (2)
        cout << "あなたはもてません" << endl;
else
        cout << "あなたはもてる可能性があります" << endl;
```

この場合は，（もっともらしく）身長が170cm以上でかつ体重が80kg以下なら，もてる可能性がある．

3.2◆どちらがもてる？

どちらが正しいかを次のサンプルプログラムでテストする．

【サンプルプログラム】

```
#include <iostream.h>

int main()
{
    int hight, weight;

    cout << "身長(cm) ? ";
    cin >> hight;
    cout << "体重(kg) ? ";
    cin >> weight;

    if (hight >= 170)
        if (weight > 80)
            cout << "あなたはもてません" << endl;
        else
            cout << "あなたはもてる可能性があります" << endl;

    return 0;
}
```

【実行例】

```
% a.out ―――――――――――――― 身長が170cmよりも低い場合
身長(cm) ? 160
体重(kg) ? 60
% a.out ―――――――――――――― 身長が170cm以上，体重が80kg以下の場合
身長(cm) ? 180
体重(kg) ? 77
あなたはもてる可能性があります
% a.out ―――――――――――――― 身長が170cm以上，体重が80kgを超える場合
身長(cm) ? 180
体重(kg) ? 85
あなたはもてません
% _
```

この結果を見ると，身長が170cmよりも低い場合は何も表示されていない．すなわち，前者ではない．また，身長が170cm以上，体重が80kg以下の場合，もてる可能性があるといっている．すなわち，後者に一致する．

以上の結果から，else節は（2）のif文に対応しているのがわかる．このように，else節は，最も近いif文に対応する．

3.3◆{} の利用

if文をネストすると，ifとelseの対応がわかりづらくなるので，{} を使って対応を明確にするとよい．たとえば（1）のif文と対応させたければ次のように書く．

```
if (hight >= 170) { ―――――――――――――― (1)
    if (weight > 80) ―――――――――――――― (2)
        cout << "あなたはもてません" << endl;
}
else
        cout << "あなたはもてる可能性があります" << endl;
```

また，（2）のif文と対応させたければ次のように書く．

```
if (hight >= 170) { ―――――――――――――― (1)
    if (weight > 80) ―――――――――――――― (2)
        cout << "あなたはもてません" << endl;
```

```
        else
            cout << "あなたはもてる可能性があります" << endl;
    }
```

4 三項演算

if文やswitch文のほかに，**三項演算**を使っても場合分けをすることができる．

三項演算は，if文やswitch文と比べて異質である．なぜなら，if文やswitch文が**文**であるのに対して，三項演算は**式**だからである．三項演算は，（式ゆえに）文の一部として使われる．

4.1◆三項演算子 ? と :

三項演算は，三項演算子 ? と : を使って記述する．そして，その名のとおり3つのオペランドで構成する．

```
式1 ? 式2 : 式3
```

式1の値が真なら**式2**が全体の値になる．また，**式1**の値が偽なら**式3**が全体の値になる．

式1が条件式になるので，ifやswitchに合わせて**式1**を（）で囲んで書くことが多い．もちろん，囲まなくても文法的には間違いではない．

```
(式1) ? 式2 : 式3
```

この式を（）で囲むと視覚的にわかりやすい

4.2◆代入への利用

三項演算自体は1つの式なので，次のように代入式の右辺に使用されることが多い．

```
val = (式1) ? 式2 : 式3
```

式1の真偽によって式2か式3のいずれかが変数valに代入される

三項演算の欠点は，プログラムがわかりにくくなることである．三項演算自体をif文に置き換えることは可能なので，無理して三項演算を使う必要はない．たとえば，いまの代入文なら，if文を使って次のように書くこともできる．

```
if (式1)
    val = 式2;
else
    val = 式3;
```

4.3◆大小を比較するプログラム（三項演算子版）

次のサンプルプログラムは，第5章第5.2節で紹介した大小を比較するプログラムを，三項演算子を使って書き換えたものである．

【サンプルプログラム】

```
// max.C -- 三項演算子版

#include <iostream.h>

int main()
{
    int num1, num2;

    cout << "2つの数の大小比較をします" << endl
         << "1つ目の数? ";
    cin >> num1;
    cout << "2つ目の数? ";
    cin >> num2;

    cout << ((num1 > num2) ? num1 : num2)
         << "のほうが大きいです" << endl;

    return 0;
}
```

【実行例】

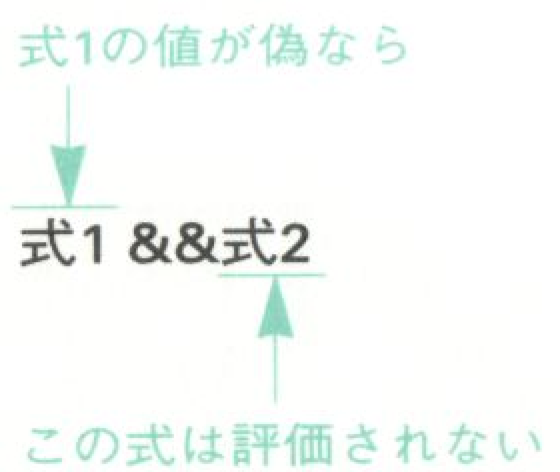
```
% a.out
2つの数の大小比較をします
1つ目の数? 100
2つ目の数? 101
101のほうが大きいです
% _
```

5 論理演算子のトリック(猛者になるために)

最後に，論理演算子を使った選択文を紹介する（少し難しい）.

&&や||は，左オペランド（左側のオペランド）を評価しただけで演算結果を決定できるときは，右オペランドを評価しない（難しい表現！）.

たとえば，&&は左右のオペランドがともに真でなければ，式全体が真にならない．したがって，左オペランドが偽なら，右オペランドを評価しなくても式全体の値を偽と決定することができる．このようなとき，C++コンパイラは右オペランドの評価を省略する.



このこと自体は妥当である．しかし，この事実を逆手に取って，トリッキーなプログラムを書く猛者がいる.

```
num1 && (num2 = 100);
```

このプログラムの意味がわかるだろうか？

&&が使われているので論理積である．まず左オペランドのnum1が評価される．これはnum1が0かどうかで評価が決まる．0なら偽である．0以外なら真である.

で，仮にnum1が0であったとする．すると左オペランドが偽なので，もはやそれ以上のことは行われない．式全体の値が偽となるが，その値は捨てられる.

一方，num2が0でなければどうか？　まだ式全体の真偽が決まらないので，&&の右オペランドが評価される．右オペランドは代入式である．代入式の値は，代入された値である．すなわち，num2に100が代入され，100が右オペランドの値となる．これは，非ゼロなので，右オペランドは真となる.

ここで，num1が0なら代入が行われず，0なら代入が行われたことに注意．つまり，このプログラムは，次のif文と同じ機能をする.

```
if (num1 != 0)      num1が0でなければ
    num2 = 100;     num2に100を代入する
```

&&を論理積ではなく，条件の選択に使用している．とんでもないプログラムである．しかし，動くことは動く．

同様に，次のようなプログラムを作ることもできる．

```
num1 || (num2 = 100);
```

この場合は，次のif文と同じである．

```
if (num1 == 0)      num1が0ならば
    num2 = 100;     num2に100を代入する
```

敢えてこのようなプログラムを書く必要はないが，他人の書いたプログラムを読むときには必要になるかもしれない．

第10章 反復文

この章では，制御文の1つである**反復文**を説明する．反復文には，以下の3種類がある．

while文
do～while文
for文

反復文を使うと，特定の文を繰り返し実行できる．これにより，すべての基本三構造が出揃うことになる．

1 インクリメント／デクリメント

反復文を使うとき，ある変数の値を1大きくしたり，1小さくするといった処理が必要になることがある．この節では，変数の値を，現在の値を基準に**相対的に増減**する方法を説明する．

1.1◆代入演算子

いま，変数xの値を+1したいとする．どのように書けばよいだろう？

```
x = x + 1;
```

とりあえず正解である．われわれのいままでの知識からすれば，この代入により変数xの値を+1することができる．1つ注意すれば，「なんでxとx + 1が等しいの

か？」と思わないこと．C++の=は，代入であって等値ではない．

で，C++では，これを次のように略記することができる．

```
x += 1;
```

+=は**代入演算子**の1つで，左辺の変数を右辺の値だけ大きくする．以下にC++に用意されている代入演算子を列挙する．

<table>
<tr><th>演算子</th><th>演算</th></tr>
<tr><td>x = val</td><td>変数xに値valを代入する
（これが代入の基本）</td></tr>
<tr><td>x += val</td><td>x = x + val と同じ</td></tr>
<tr><td>x -= val</td><td>x = x - val と同じ</td></tr>
<tr><td>x *= val</td><td>x = x * val と同じ</td></tr>
<tr><td>x /= val</td><td>x = x / val と同じ</td></tr>
<tr><td>x %= val</td><td>x = x % val と同じ</td></tr>
<tr><td>x >>= val</td><td>x = x >> val と同じ</td></tr>
<tr><td>x <<= val</td><td>x = x << val と同じ</td></tr>
<tr><td>x &= val</td><td>x = x & val と同じ</td></tr>
<tr><td>x |= val</td><td>x = x | val と同じ</td></tr>
<tr><td>x ^= val</td><td>x = x ^ val と同じ</td></tr>
</table>

注意　この表には，まだ説明していない演算子が含まれている．

次に，いくつか例を示す．

```
x += 100;   ―― xを100大きくする
x -= 100;   ―― xを100小さくする
x += -100;  ―― 上と同じ
x %= 4;     ―― xを4で割った余りをxに代入する
```

1.2◆インクリメント／デクリメント演算子

+-する値が1のときは，+=や-=を使うよりも，さらに便利な演算子が用意されている．

演算子	演算
++x	変数xの値を+1する 式の値は+1を実行した後のxの値 x += 1 と同じ
x++	変数xの値を+1する 式の値は+1を実行する前のxの値
--x	変数xの値を-1する 式の値は-1を実行した後のxの値 x -= 1 と同じ
x--	変数xの値を-1する 式の値は-1を実行する前のxの値

++，--ともに単項演算子である．オペランドは変数が1個である．そして，その変数の値を+1または-1する．

++や--を変数の前に付ける場合を**前置インクリメント演算子**，**前置デクリメント演算子**という．その場合，式全体の値は，+1や-1を実行した後の変数の値になる．

また，++や--を変数の後ろに付ける場合を**後置インクリメント演算子**，**後置デクリメント演算子**という．その場合，式全体の値は，+1や-1を実行する前の変数の値になる．

以下に，これらの演算の例を表にまとめて示す．

演算	演算後のx	演算後のy
x = 1; y = (++x);	2	2
x = 1; y = (x++);	2	1

2 while

この節では，猫と戯れながら，最初の反復文としてwhile文を説明する．

```
while (条件式)
    文
```

文 ← 条件式が真なら，この文が繰り返し実行される
偽なら，この文は実行されない

while文は（）の中の条件式が真の間，直後の文を繰り返し実行する．条件式の書き方は，if文と同じである．2行目にある繰り返し文は，4文字分インデントして書く．

2.1◆無限ループ

while文の例を示す．

```
while (0)
    cout << "猫ふんじゃった" << endl;
```

この場合，条件式は0なので偽である．したがってcoutで始まる文は実行されない．

```
while (1)
    cout << "猫ふんじゃった" << endl;
```

今度は，条件式が1なので真である．したがって，coutで始まる文が繰り返し実行される．実際，このプログラムを実行すると大変なことになる．次に，これをコンパイル可能なプログラムに仕上げたものを示す．

【サンプルプログラム】

これは，while文をテストするものである．ただし，実行すると無限ループに入るので，実行しないほうがよい．

```
// neko.C(第1版) -- while文のテスト

#include <iostream.h>

int main()
{
    while (1)
        cout << "猫ふんじゃった" << endl;

    return 0;
}
```

次に，このプログラムの実行例を示す（真似しないほうがよい）．

【実行例】

```
% a.out
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
.........................
.........................

% _ ―――――――― Ctrl + C で割り込みをかけて止める
```

“猫ふんじゃった”を無限に表示し続ける．当然である．while文は，条件式が真の間は，文を繰り返し実行するのだから….

2.2◆ループ変数

このように見てくると，while文はあまり実用性がないように見える．なにせ，条件式の値によって，文をまったく実行しないか，あるいは永遠に実行するかのいずれかになってしまうのだから….

で，実際はwhile文をこのように使うことは少ない．通常は，繰り返しを制御するための変数（これを**ループ変数**という）を用意し，その値を繰り返しの中（これを**ループの本体**という）で変化させる．そして，それを条件式の判定に使う．こうすれば無限ループに陥ることなく，while文を実用的に使うことができる．

簡単な例から始める．まずループ変数を用意する．ループ変数にはi，j，kなどを使うことが多い．これは，古き時代のFORTRANからの影響である．

```
int i = 7; ―――――――― ループ変数
                      初期値は繰り返しを行う回数
```

そして，ループ変数を条件式の中に入れる．

```
while (i) ―――――――― ループ変数の値によって繰り返しを
                     行うかを判断する
```

そして，最後にループの本体を書く．

```
while (i)
    cout << "猫ふんじゃった" << endl;
```

ただし，これではループ変数の導入前と変わらない．無限ループになってしまう．繰り返しを有限回でストップさせるには，ループの本体の中でループ変数の値を変化させればよい．この場合，ループの本体が複数になるので，if文でも使用したブロックを導入する．

```
while (i) {  ――― 複数の文を置くためにブロックを導入
    cout << "猫ふんじゃった" << endl;
}
```

そして，ブロックの中に --i（ループ変数の値を1つ小さくする）を置く．

```
while (i) {
    cout << "猫ふんじゃった" << endl;
    --i;  ――― ここでループ変数の値をいじる
}
```

これで，ループの本体が実行されるたびに，ループ変数iの値が1つずつ小さくなる．やがて0になると，偽と判定されてwhile文を終了する．

条件判断	ループ変数の値	ループ本体の実行
第1回目	7（真）	実行される
第2回目	6（真）	実行される
第3回目	5（真）	実行される
第4回目	4（真）	実行される
第5回目	3（真）	実行される
第6回目	2（真）	実行される
第7回目	1（真）	実行される
第8回目	0（偽）	実行されない while文を終了する

この結果，ループ変数の初期値と同じ7回分，ループの本体が繰り返し実行される．以上をまとめると，次のようになる．

```
int i = val; // valは繰り返しを行う回数
while (i) {
```

```
        // ここに繰り返しを行う処理を書く
        --i;
    }
```

これが，while文の基本的な使い方である．

2.3◆猫ふんじゃった

次に“猫ふんじゃった”を無限ループにならないように改良したものを示す．

【サンプルプログラム】

これは，“猫ふんじゃった”を7回繰り返すプログラムである．

```
// neko.C(第2版) -- while文のテスト
//                  今度は無限ループにならない

#include <iostream.h>

int main()
{
    int i = 7;

    while (i) {
        cout << "猫ふんじゃった" << endl;
        --i;
    }

    return 0;
}
```

【実行例】

```
% a.out
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
% _
```

3 定数

2つ目の反復文に入る前に，**定数**について説明する．

次は一般的な変数宣言である．宣言と同時に初期値を与えている．

```
int x = 100;
```

次は，この変数宣言の前に**const**というキーワードを付けたものである．

```
const int x = 100;
```

このように変数宣言の前にconstを付けると，xは変数ではなく，**定数**になる．

定数は変数と違って，値を変更することができない．たとえば次のようなことをしようとすると，コンパイルエラーになる．

```
x = 200;  ――――― 定数に別の値を代入しようとしている
++x;      ――――― これも定数の値を変更しようとしているのでダメ
```

定数には新たな値を代入できないので，宣言と同時に初期設定しなければならない．したがって，次のような定数宣言は，コンパイルエラーになる．

```
const int x;  ――――― 初期値を与えていない
```

定数は，次のような場面で使われる．

- **プログラムの中で変わらない値を表現する**
- **値を変更されては困る変数**

たとえば，次は円周率を定数として宣言している．

```
const float PI = 3.14159;
```

このように，定数に使用する名前には大文字を使うことが多い．

以前，「いろいろなデータ型」のところで100のような具体的な値のことを定数ではなくリテラルと呼ぶことを説明した．これは，constで宣言された定数と区別するためである．

リテラル ――――― 具体的な値
定数 ――――― constを付けて宣言された変数

4 do～while

while文は，繰り返しの先頭で条件を判断する．しかし，プログラムによっては繰り返しの最後で条件を判断したいこともある．その場合は，while文の代わりに**do～while文**を使う．

4.1◆書式

次にdo～while文の書式を示す．

```
do
    文
while (条件式);
```

文 ――――― この文が繰り返される

「文」の部分を4文字インデントする．この書式によって，条件式が真の間，doとwhileに挟まれた文が繰り返し実行される．偽なら実行されない．

4.2◆do～while文とwhile文

do～while文は，書式がwhile文に似ている．とりわけwhileで始まる行は，そっくりである．

while (条件式) ――――― while文の先頭
while (条件式); ――――― do～while文の最後

違いは，セミコロンである．do～while文の場合，最後にセミコロンが付く．このようにdo～while文とwhile文は書式が似ているので，次のように文の部分を{}で囲ってブロック化することが多い（たとえ繰り返す文が1つであっても）．

```
do {
    文
} while (条件);
```

こうすると先頭に}が付くので，while文と見分けがつけやすくなる．

4.3◆猫ふんじゃった（do～while版）

次にwhile文のところで作成した“猫ふんじゃった”をdo～while文で書き換えたものを示す．

【サンプルプログラム】

```
// neko.C(第3版) -- do～while文のテスト

#include <iostream.h>

int main()
{
    int i = 7;

    do {
        cout << "猫ふんじゃった" << endl;
        --i;
    } while (i);

    return 0;
}
```

【実行例】

```
% a.out
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
% _
```

4.4◆入力の検査

“猫ふんじゃった”を見ているかぎり，do～while文もwhile文もあまり変わりがないように見える．

do～while文がwhile文と異なるのは，ループの本体が少なくとも1回は実行されるということである．while文の場合は，条件式が最初から偽になっていればループの本体は1回も実行されない．これがwhile文と大きく違う点である．

この特性を活かして，入力の検査にdo～while文がよく使われる．

```
do {
    cin >> x;  ──────── 正しい入力が得られるまで
                        入力を繰り返す
} while (…);
```

最低1回は入力をしなければならないので，この処理はwhile文よりもdo～while文のほうが向いている．もし，これをwhile文で書こうとすると，次のようにcinを2回書かなければならなくなる．

```
cin >> x;  ──────── ここと
while (xが正しくない) {
    エラーメッセージ
    cin >> x;  ──── ここ
}
```

4.5◆円の面積と円周

次に，入力の検査を行うサンプルとして「円の面積と円周」を求めるプログラムを示す．

プログラムの中で半径を入力する．半径は負ではまずい．そこで，入力された半径が負でないかを検査し，正しい値が入力されるまでdo～while文で入力を繰り返すということをやっている．

【サンプルプログラム】

これは，「円の面積と円周」を求めるプログラムである．do～while文による入力の検査のサンプルプログラムになっている．

```
// circle.C -- 円の面積と円周

#include <iostream.h>

int main()
```

```
{
    float r;
    const float PI = 3.14159;

    cout << "円の面積と円周を求めます" << endl
         << "最初に半径を入力してください" << endl
         << "ただしマイナスでは計算できません" << endl
         << endl;

    do {
        cout << "半径? ";
        cin >> r;
    } while (r < 0);

    cout << endl
         << "面積 = " << PI * r * r << endl
         << "円周 = " << 2 * PI * r << endl;

    return 0;
}
```

【実行例】

```
% a.out
円の面積と円周を求めます
最初に半径を入力してください
ただしマイナスでは計算できません

半径? -10 ―――――――――――― マイナスを入力
半径? 10 ―――――――――――― 今度は適正な値を入力

面積 = 314.159
円周 = 62.8318
% _
```

5 for

3つ目の反復文は，**for文**である．for文は，while文を拡張したものと考えることができる．

5.1◆while文の拡張

while文を使うとき，無限ループになるのを避けるためループ変数を用意することが多い．次のように．

```
int i =繰り返しの回数; ――――――― ループ変数の初期化
while (条件式) { ――――――――――― while文
    反復文の並び
    ループ変数の更新;
}
```

このようにwhile文は，直前でループ変数を初期化し，ループ本体の最後でループ変数の更新を行うことが多い．それを1つの文ですべてまかなえるようにしたのがfor文である．

5.2◆書式

for文の書式を次に示す．

```
for (式1; 式2; 式3)
    文
```

このfor文は次のwhile文と等値である．

```
式1;
while (式2) {
    文
    式3;
}
```

5.3◆for文の動作

for文は，次のように実行される．

① まず「式1」が評価される

式1には，どんな式でも置くことができる．また，省略することもできる．しかし，通常はループ変数の宣言とその初期化を行うのが普通である．次のように．

```
for (int i = 0; …)
```

（↑ 式1）

> **注意**　ここで「式1」のところで変数宣言を行っていることに注意.
> 　変数宣言は文なので，本来なら「式1」のところで変数宣言を行うことはできない. 実際，C言語では変数宣言はブロックの先頭に限られていた. しかし，C++はfor文の「式1」のところで変数宣言を行うことができる. また，文が置けるところならどこでも変数宣言を行うことができる.

② 続いて「式2」を条件式として真偽を判定する

式2はwhile文の（）の中に書く条件式に相当する. ここに，繰り返しを行うかどうかの判定をするための式を書く. 次のように.

```
for (…; i < 7; …)
```

（↑ 式2）

③ その結果が真ならループの本体が実行される

④ 最後に「式3」が実行される

式3にも，どんな式でも置くことができる. また，省略することもできる. しかし，通常はループ変数の更新を行うのが普通である. 次のように.

```
for (…; ; i++)
```

（↑ 式3）

⑤ 再び②に戻る

以上の動作を，②で偽と判定されるまで繰り返す.

5.4◆猫ふんじゃった（for版）

for文は，あらかじめ繰り返しの回数がわかっている場合に使うと便利である. たとえば，ループの本体を7回繰り返したいときは，次のように書く.

for (int i = 0; i < 7; i++)
　　文

次にdo〜while文のところで作成した“猫ふんじゃった”をfor文で書き換えたものを示す．

【サンプルプログラム】

```
// neko.C(第4版) -- for文のテスト

#include <iostream.h>

int main()
{
    for (int i = 0; i < 7; i++)
        cout << "猫ふんじゃった" << endl;

    return 0;
}
```

【実行例】

```
% a.out
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
猫ふんじゃった
% _
```

3つの反復文による“猫ふんじゃった”では，for文によるものが最もコンパクトであることがわかる．

6 ループ変数の利用

for文では，ループ変数の値をループの本体の中で参照することがある．たとえば次は，変数iをループ変数に使用している．

for (int i = 1; i <= 10; i++)

この場合，ループ変数は繰り返しを実行するたびに，次のように変化する．

1→2→3→ … →10

したがって，ループの本体でiを参照することにより，現在の繰り返しの回数を知ることができる．これを利用すると，現在の繰り返し回数を表示するプログラムを作ることができる．

6.1◆繰り返しの回数

次は，ループ変数を参照するサンプルプログラムである．

【サンプルプログラム】

このプログラムは，現在の繰り返し回数を表示する．

```
// repeat.C -- 繰り返しの数を表示する

#include <iostream.h>

int main()
{
    for (int i = 1; i <= 10; i++)
        cout << "現在" << i << "回目の繰り返し中です"
             << endl;

    return 0;
}
```

【実行例】

```
% a.out
現在1回目の繰り返し中です
現在2回目の繰り返し中です
現在3回目の繰り返し中です
現在4回目の繰り返し中です
現在5回目の繰り返し中です
現在6回目の繰り返し中です
現在7回目の繰り返し中です
現在8回目の繰り返し中です
現在9回目の繰り返し中です
現在10回目の繰り返し中です
% _
```

6.2◆平方表

ループ変数の参照は，各種数表を作成するのにも利用できる．たとえば，次は，1から10までの数の平方を表にして出力するプログラムである．

【サンプルプログラム】

```
// square.C -- 平方表

#include <iostream.h>

int main()
{
    for (int i = 1; i <= 10; i++)
        cout << i << " " << i * i << endl;

    return 0;
}
```

【実行例】

```
% a.out
1 1
2 4
3 9
4 16 ———— この行は4の平方が16であることを示している
5 25
6 36
7 49
8 64
9 81
10 100 ———— ここで桁数がずれている
% _
```

これで一応，数表としては完成しているが，桁数によって表示のずれが生じて見にくい．次に，これをもう少し見やすい表に改良する．そのためには，coutによる出力桁数を制御する必要がある．それを次の章で説明する．

第11章 高度なストリームI/O

ストリームI/Oの基本については，第3章から始まる各章で説明した．この章では，やや高度なストリームI/Oを説明する．そして，残された課題——すなわち，

- 不正な文字列が入力されたら，プロンプトを表示して再入力を促す
- 演算数が負数の場合は（）を付けて表示する

をクリアする．

1 出力幅の制御

この節では，coutによる出力幅を制御する方法を説明する．

1.1◆setw

coutでは，**マニピュレータ**という特別な記号を使用することができる．マニピュレータの例としては，すでにendlやhex，decを知っている．

そのマニピュレータの中に**setw**というのがある．このマニピュレータを使うことによって，出力幅を制御することができる．setwは次のようにして使う．

```
cout >> … >> setw(出力幅) >> …
```

たとえば，次のようにすると，変数iを7文字分の幅で出力することができる．

```
cout << setw(7) << i << endl;
```

出力幅を7に設定する

この場合，変数iの値が15なら次のように出力される．

■■■■■15（■はスペースを表す）

7文字分の幅

ここで■は，半角のスペースで，実際は目に見えない．15は2桁しかないので，左に5個のスペースが補われて，ちょうど7桁になるように整形される．

逆に変数iの値が123456789のように7桁を超える場合は，ちゃんと123456789が，そのまま出力される．7桁に切り詰められることはない．

注意　左に補う文字をスペース以外の文字に変更することもできる．

1.2◆iomanip.h

マニピュレータsetwを使うとき，注意しなければならない点が2つある．

1つは，setwを使うときには**iomanip.h**が必要になるということである．普通，coutを使うときには次の1行が必要である．

```
#include <iostream.h>
```

setwを使うときには，次の1行も必要になる．さもないとコンパイルエラーになる．

```
#include <iomanip.h>
```

つまり，プログラムの先頭に次の2行を書く．

```
#include <iostream.h>
#include <iomanip.h>
```

ただし，iomanip.hは内部でiostream.hをinclude（インクルード）しているので，iomanipをincludeするときは，iostream.hのincludeを省略してもかまわない．つまり，次の1行を書くだけでよい．

```
#include <iomanip.h>
```

1.3◆有効な範囲

マニピュレータsetwを使うとき，注意しなければならないもう1つの点は，setwによる出力幅の制御が直後の出力にしか影響を与えないということである．

たとえば変数xと変数yの値をともに8桁で出力したいとする．しかし，次のようにしてもダメである．

```
cout << setw(8) << x << y
```

setwは直後のxにしか影響を与えない．変数yも8桁で出力したいなら，yの前にもsetwを置く必要がある．

```
cout << setw(8) << x << setw(8) << y
```

2箇所必要

2 平方表（改良版）

coutによる出力幅の制御ができるようになったので，第10章の第6.2節で作成した平方表の出力を改良する．改良点は，次の2点である．

- 1桁の1～9，および2桁の10で表示がずれないようにする
- ヘッダを追加して表を見やすくする

【サンプルプログラム】

このサンプルプログラムは，1から10までの数の平方数を表にして出力するものである．setwを使って出力幅を制御している．

```
// square.C(第2版) -- 平方表
//                    setwによる書式の設定

#include <iomanip.h>

int main()
```

```
{
    cout << "  x   x * x" << endl
         << "-----------" << endl;

    for (int i = 1; i <= 10; i++)
      cout << setw(3) << i
           << setw(8) << i * i << endl;

    return 0;
}
```

【実行例】

```
% a.out
  x   x * x ―――――――――――――― ヘッダ
-----------
  1       1
  2       4
  3       9
  4      16
  5      25
  6      36
  7      49
  8      64
  9      81
 10     100 ―――――――――――――― 表示がずれていない
% _
```

3 cinのエラー処理

この節では，第7章の「cinの落とし穴」で指摘しておいた課題に取り組む．また，具体的なプログラムの改良を，次の第4節で行う．

3.1◆混乱するcin

第7章の「cinの落とし穴」で指摘しておいた課題は2つある．1つは，型の違うデータが入力されたとき，cinが混乱するというものである．すなわち，

```
int num1, num2;
```

のようにint型の変数が2つあって，

```
cin >> num1;
cin >> num2;
```

のように，それぞれの変数にデータを入力しようとしたとき，intではなく文字列が入力されると，その後のcinの挙動がおかしくなるというものである．そのときの実行例を次に再掲する．

```
% a.out
10進数(被演算数)？Hello ──── 数ではなく文字列を入力
10進数(演算数)？0 + 0 = 0 ──── 何も入力しないのに表示される
% _
```

もう1つの課題は，演算数に負数が入力された場合，() を付けて表示するというものである．

【改良前】 10 + -7 = 3
【改良後】 10 + (-7) = 3

この課題の改良は，第7項で行う．

3.2◆エラーフラグ変数

cinは，内部にstateという変数を持っている．そして，入力中にエラーが発生すると，エラーの状態を，この内部変数に記録するようになっている．本書では，stateを**エラーフラグ変数**と呼ぶことにする．

エラーフラグ変数に記録される状態は，次の3種類である．

ファイルの終わりに達している
回復可能なエラーが発生している
回復不能なエラーが発生している

注意　処理系によっては，これ以外のエラーを扱うものもある．

cinで型の違うデータが入力されると，もちろんそのエラーがエラーフラグ変数に記録される．エラーが発生していると，cinから正常なデータを入力できない．

そこで，データを入力したら，エラーが発生しているかどうかを調べ，可能ならエラーを回復し，エラーフラグ変数をクリアしてやれば，cinの混乱を回避することができる．

3.3◆演算子！

cinでエラーが発生しているかどうかを調べるのは，実はかなり難しい．しかし，幸いcinは！という演算子を再定義していて，!cinで簡単にエラーの有無を検出できるようになっている．

```
if (!cin)
    // cinでエラーが発生している
```

したがって，不正な入力をチェックして再入力させるには，次のようにすればよい．

```
cin >> num1;          ——— cinで入力
while (!cin)          ——— エラーが発生していれば
    cin >> num1;      ——— 入力をやり直す
```

ただし，これでは少し具合が悪い．なぜなら，cinで一度（エラーフラグ変数に）エラーが記録されると，それを解除するまでエラー状態が続く．したがって，上のようなコードを書くと，エラーが発生したときに無限に入力を続けなければならなくなってしまう．この状況を改善するためには，while文の中でエラーをリセットする必要がある．

```
cin >> num1;
while (!cin) {        ——— エラーが発生していれば
    // エラーをリセット
    cin >> num1;      ——— 入力をやり直す
}
```

3.4◆エラーフラグのリセット

cinのエラーフラグをリセットするには，clear関数を使う．

```
cin.clear()
```

したがって，次のようにwhileの本体でこの関数を使えば，入力を繰り返すこと

はなくなる（はずである）．

```
cin >> num1;
while (!cin) {
    cin.clear();  ←―――― 追加
    cin >> num1;
}
```

しかし，残念ながら，これだけではまだ不足である．やはり，エラーが発生すると，無限に入力を続けなければならなくなってしまう．

この原因は，不正に入力された文字がcinの入力バッファに残ってしまうためである．バッファ中の入力文字は消費されない．したがって（たとえエラーフラグをクリアしようとも），次の入力で同じ文字を入力しようとして再びエラーになってしまう．

この状況を改善するためには，エラーフラグの解除とともに，バッファに残された不正な文字を捨てる必要がある．

3.5◆入力バッファのクリア

cinの入力バッファに残された文字をクリアするには，ignore関数を使う．

```
cin.ignore()
```

したがって，次のようにcin.clearと対にしてこの関数を使えば，入力を繰り返すことがなくなる．

```
cin >> num1;
while (!cin) {
    cin.clear();
    cin.ignore();  ←―――― 追加
    cin >> num1;
}
```

これで，次の入力ができなくなることはない．しかし，まだ（ほんのわずかだが）問題が残っている．

3.6◆プロンプトの嵐

ここまでで，cinによる不正な入力をクリアし，再入力できるようになっている．しかし，再入力のときにその旨を示すプロンプトを出力するようにすると，少し問題が出る．

```
cin >> num1;
while (!cin) {
      cin.clear();
      cin.ignore();
      cout << "不正な入力です" << endl ◀――― 追加
      << "正しい整数? ";
      cin >> num1;
}
```

この追加により，不正な入力をすると，次のように入力を促すプロンプトが出力されるようになる．

```
% a.out ――――――――― プログラムの実行開始
H ――――――――――――― 整数ではなく文字を入力
不正な入力です ―――――― プロンプトが出力される（これはよい）
正しい整数?
```

この場合は，Hのように（不正な文字が）1文字だけだったので，これで済んだ．しかし，次のように複数の文字を入力すると，その文字数だけプロンプトが出力されてしまう．

```
% a.out
Hello ――――――――――――― 5文字の不正な文字を入力
不正な入力です ―――――――― プロンプトが5回出力される
正しい整数?不正な入力です
正しい整数?不正な入力です
正しい整数?不正な入力です
正しい整数?不正な入力です
正しい整数?
```

これは，cin.ignoreがバッファから1回につき1文字だけ取り除くからである．したがって，すべての文字を取り除くまでwhileの本体が実行されてしまう．

この状況を改善するためには，バッファに残された不正な文字列をすべて取り除く必要がある．

3.7◆INT_MAX

ignoreは，引数を指定することで複数の文字をバッファから削除することができる．

```
cin.ignore(文字数);
```

問題は，何文字指定すればよいかである．ユーザーが不正な文字を何文字入力するかは，プログラムを実行するまでわからない．そこで，指定可能な文字数の最大値を指定してやる．

INT_MAXという記号定数を使うと，整数の最大値を指定することができる．

```
cin.ignore(INT_MAX);
```

これなら，確実にすべての文字を捨てることができる．ただし，INT_MAXを指定するためには，limits.hをincludeする必要がある．すなわち，プログラムの先頭部分に次の1行を追加する．

```
#include <limits.h>
```

3.8◆複数文字のクリア

cin.ignore（INT_MAX）は，いかにも大袈裟である．まさか，整数の最大値分もの数の不正な文字が入力されることはない．

ignoureは，さらに2つ目の引数を指定することができる．

```
cin.ignore(文字数,打ち切り文字);
```

第2引数は，**打ち切り文字**である．

たとえば打ち切り文字に“w”を指定したとする．すると，ignoreは第1引数で指定された文字数分の不正文字をバッファから削除しようとする．しかし，その途中で第2引数で指定された文字が現れると，そこで削除を打ち止めにする．

バッファ：**Hello, world**

↑ ここでクリアを打ち切る

この結果，バッファには“w”より後ろの文字が残ることになる．

バッファ：orld

この打ち切り文字の機能を利用して，次のように指定する．

cin.ignore(INT_MAX, '\n'); ――――(1)

ここで“\n”は改行文字を表す．cinでキーボードからデータを入力すると，最後に必ず“\n”が入る．したがって，(1) のように指定すれば，不正な文字をすべて削除した後，そこでクリアを打ち切ることができる．INT_MAX文字分のクリアを行うことはない．

4 2数の合計（改良版）

cinのエラー処理の応用として，第7章の「cinの落とし穴」で指摘しておいた課題に取り組む．単純な2数の加算という問題でありながら，「問題あり！」とされたプログラムの改良である．今回の改良点は，次の2点である．

- 不正な文字列が入力されたら，プロンプトを表示して再入力を促す
- 演算数が負数の場合は（）を付けて表示する

最初に改良版のプログラムを示し，上の2点が改良されたことを確認した後，プログラムのポイントを説明する．

4.1◆リスト

【サンプルプログラム】

これは，第7章で示した2数の合計を求めるプログラムの改良版である．

```
// add.C -- 2数の加算

#include <iostream.h>
#include <limits.h>    // for INT_MAX

int main()
{
    int num1, num2;
```

```
    // 被演算数の入力
    cout << "10進数(被演算数) ? ";
    cin >> num1;
    while (!cin) {
        cin.clear();
        cin.ignore(INT_MAX, '\n');
        cout << "正しい10進数ではありません" << endl
             << "10進数(被演算数) ? ";
        cin >> num1;
    }

    // 演算数の入力
    cout << "10進数(演算数) ? ";
    cin >> num2;
    while (!cin) {
        cin.clear();
        cin.ignore(INT_MAX, '\n');
        cout << "正しい10進数ではありません" << endl
             << "10進数(演算数) ? ";
        cin >> num2;
    } while (!cin);

    // 結果の出力
    if (num2 >= 0)
        cout << num1 << " + " << num2 << " = "
             << num1 + num2 << endl;
    else
        cout << num1 << " + (" << num2 << ") = "
             << num1 + num2 << endl;

    return 0;
}
```

4.2◆実行

正しく改良されたか，実際に実行して確かめる．最初は，不正な文字列が入力された場合の処理である．

【実行例（その1)】

```
% a.out
10進数(被演算数) ? Hello ――――― 不正な文字列
正しい10進数ではありません ――――― プロンプトは1回
10進数(被演算数) ? 427 ――――― 入力をやり直す
10進数(演算数) ? 51
427 + 51 = 478 ――――― 正しく計算されている
% _
```

続いて，演算数に負数が入力された場合である．

【実行例（その2)】

```
% a.out
10進数(被演算数) ? 35
10進数(演算数) ? -12 ――――― 負数を入力
35 + (-12) = 23 ――――― () が付いている
% _
```

いずれも正しく機能している．

4.3◆プログラムのポイント

改良版プログラムにおけるポイントを説明する．

●不正な入力の検査

まず，cinで入力するたびに不正な入力かどうかを検査している．そして，不正なら再入力を促している．たとえば被演算数の入力なら，次のように．

```
cout << "10進数(被演算数) ? ";
cin >> num1;
while (!cin) { ――――― 不正な入力なら再入力
    cin.clear();
    cin.ignore(INT_MAX, '\n');
    cout << "正しい10進数ではありません" << endl
         << "10進数(被演算数) ? ";
    cin >> num1;
}
```

このとき，エラーフラグをクリアするため，次の2つの関数を使っていることに注意.

```
cin.clear();
cin.ignore(INT_MAX, '\n');
```

また，INT_MAXを使うためにlimits.hをincludeしていることにも注意.

```
#include <limits.h>        // for INT_MAX
```

●演算数が負数の場合

演算数に負数が入力された場合に（）を付けて表示する部分は，次のif文で処理している.

```
if (num2 >= 0) ──────────────── 負数かどうかテスト
    cout << num1 << " + " << num2 << " = "
         << num1 + num2 << endl;
else ──────────────────────── 負数なら（）を付ける
    cout << num1 << " + (" << num2 << ") = "
         << num1 + num2 << endl;
```

第12章 関数

C++への最初の旅を始めるとき，「C++のプログラムは，関数を列挙したものである」ということを説明した．

【C++のプログラム】
関数1
関数2
関数3
.....

しかし，われわれは，まだmainという名前のただ1つの関数しか使っていない．小さなプログラムならそれでもよい．中規模程度のプログラムでも，無理をすればmain関数1つで済ませることもできるだろう．しかし，それではやがて限界がくる．

この章では，1つのプログラムを複数の関数で構成する方法を説明する．

1 関数定義

新しい関数を使うためには，その関数を定義する必要がある．main関数を定義する方法なら，すでに知っている．それ以外の関数を定義するのも，mainを定義する方法と同じである．

新しい関数を定義するには，次の各要素を定義する必要がある．

①関数の名前
②関数の使い方
　　その関数の呼び出し方
　　具体的には関数の型と引数
③その関数の処理内容
　　その関数を実行すると何が処理されるか

以下，これらの各要素をどのように定義していくかを説明していく．

1.1◆関数名

一般的な関数定義は，次のような形をしている．

```
型 関数名()
{
    文の並び
    return文
}
```

このうち「関数名」に相当する部分が①の「関数の名前」である．たとえば，次はmainという名前の関数を定義している．

```
int main() ———— 関数の名前はmain
{
    文の並び
    return文
}
```

また，次はfooという名前の関数を定義している．

```
long foo() ———— 関数の名前はfoo
{
    文の並び
    return文
}
```

1.2◆関数の型

続いて②の「関数の使い方」についてである．これにより，その関数の呼び出し方が決まる．この部分は，「関数の型」と「引数」に分かれる．

関数定義のいちばん最初に書かれる部分が「関数の型」になる．たとえば，関数mainの型はintである．

```
int main() ――――――――― 関数の型はint
{
    ............
}
```

また，関数fooの型はlongである．

```
long foo() ――――――――― 関数の型はlong
{
    ............
}
```

関数の型というのは，その関数が返す値の型のことである．一般に関数は何かを処理し，その処理結果を返す．それを**関数値**という．関数の型は，関数値の型のことである．

1.3◆関数値

関数値の具体的な値は，ブロック{}の中のreturn文で指定する．

```
型 関数名()
{
    文の並び
    return文 ――――――――― この文で関数値を定義する
}
```

これまでのプログラムでは，returnの後には，いつも0を指定してきた．

```
return 0; ――――――――― 関数値が0であることを示す
```

しかし，returnの後ろは0とは限らない．任意の値を書くことができる．たとえば，次は100を指定している．

```
return 100; ―――――― 関数値が100であることを示す
```

この100が，この関数の関数値になる．すなわち，この関数を実行すると，100という値が結果として帰ってくる．

1つ注意しなければならないのは，returnの後ろに指定する値と関数の値の型を合わせなければならないということである．たとえば，次の関数fooの型はlongである．したがって，関数値もlongの値でなければならない．

```
long foo() ―――――― 関数の型はlong
{
    文の並び
    return longの値;―――― ここで指定する値もlong
}
```

1.4◆void型関数

C++では値を返さない関数を定義することもできる．関数値のない関数である．数学から見ると，値のない関数は奇妙に見える．しかし，C++では，むしろ値なし関数のほうがポピュラーである．

値のない関数を定義するには，関数の型に**void**を指定する．

```
void 関数名()
{
    ………
}
```

void型の関数は値を返さない．すると，return文の書き方がわからなくなる．

```
return 関数値;
       ↑
ここに書く値はいくつ？
```

void型の関数は，そもそも値を返さないのだから，return文で関数値を指定するのはナンセンスである．そのためreturn文は関数値を省略する書き方を許している．

```
return; ―――――――― 値を指定しない
```

したがって，値を返さない関数の一般的な形式は次のようになる．

```
void 関数名()
{
    文の並び
    return; ―――――――― 関数値を指定しない
}
```

注意　void型の関数でreturn文に関数値を指定すると，コンパイル時に，警告エラーが出る．警告エラーの場合は，実行ファイルが作成されるので実行可能である．

void型関数の場合，関数の最後に置くreturn文は省略可能である．void型関数でreturn文が必要になるのは，関数の途中で終了する場合である．

```
void 関数名()
{
    ..........
    if (何かの条件)
        return; ◀―――――― ここ
    .......... ◀―――――― 関数の最後のreturnは不要
}
```

次にvoid型関数の例を示す．

```
void apple()
{
    cout << "func: apple running..." << endl;
    return; ―――――――――――― このreturnは省略可能
}
```

1.5◆引数

②の引数に相当する部分は，（）の中で指定する．

```
型 関数名() ――――――――― () の中に引数を指定する
{
    文の並び
    return文
}
```

引数がない場合は，() の中を空のままにしておく．たとえば，次の関数mainもfooもいずれも引数がない．

```
int main() ――――――――― 引数はない
{
    …………
}

long foo() ――――――――― 引数はない
{
    …………
}
```

引数の詳細については，第13章で説明する．

1.6◆関数内部の実行順序

③の「その関数の処理内容」は，ブロック {} の中に書く．関数のブロック内部には，main関数と同様，複数の文を置くことができる．

```
型 関数名()
{
    文1
    文2
    文3
    ………
}
```

この場合，プログラムは文が置かれた順序――すなわち「文1→ 文2→ 文3→ …」の順に実行される．これらの文が，この関数の処理内容になる．

1.7◆関数を並べる順序

1つのプログラムを複数の関数で構成する場合，関数を並べる順序は自由である．たとえばapple，orange，mainという3つの関数でプログラムを作るとする（mainは必須である）．この場合，3つの関数を，たとえば次の順序に並べることができる．

```
void apple()
{
    .........
}
void orange()
{
    .........
}
void main()
{
    .........
}
```

あるいは，次の順序で並べることもできる．

```
void orange()
{
    .........
}
void main()
{
    .........
}
void apple()
{
    .........
}
```

とにかく関数を並べる順序は自由である．しかし，関数をどのような順序で並べようと，プログラムはmainの最初の文から実行される．

注意　実は関数を並べる順序には意味がある．それについては第3節で詳しく説明する．

1.8◆サンプルプログラム

実際に1つのプログラムを複数の関数で構成する例を示す．

【サンプルプログラム】

このプログラムは，1つのプログラムがapple，orange，mainという3つの関数で構成されている．関数はapple→main→orangeの順に並べてある．

```
// funcs.C -- 複数の関数を使う

#include <iostream.h>

void apple()
{
    cout << "func: apple running..." << endl;
}

int main()
{
    cout << "func: main running..." << endl;
    return 0;
}

void orange()
{
    cout << "func: orange running..." << endl;
}
```

【実行例】

```
% a.out
func: main running...
% _
```

この実行例を見ると興味深いことがわかる．

まず，関数はapple→main→orangeの順に並べてある．にもかかわらず，実行は

関数mainの最初の文から開始されている．これは，C++プログラムの特性である．

C++のプログラムは関数mainの最初の文から実行を開始する

それと，もう1つは，関数appleとorangeに書かれた文はいずれも実行されていないということである．これは，関数mainを除き，関数は定義しただけでは使われないからである．

関数は，定義しただけでは実行されない．明示的に呼び出す必要がある．次の節では，定義した関数を呼び出す方法を説明する．

2 関数参照

関数は，定義しただけでは実行されない．この節では，定義した関数を呼び出す方法を説明する．

2.1◆関数を呼び出す

定義した関数を実行するには，**関数参照**を使う．関数参照は，関数の名前に（）を付けたものである．

関数名()

たとえば，関数appleを実行するなら次のようにする．

```
apple();
```

関数参照は式なので，式が置けるところならどこにでも置くことができる．また，セミコロンを付ければ，式文として単独の文にすることもできる．

```
apple()  ―――――― これは式
apple(); ―――――― これは文
```

2.2◆複数の関数を使う

次に前のサンプルプログラムに関数参照を追加し，appleやorangeを実行できるようにしたものを示す．

【サンプルプログラム】

```
// funcs.C(第2版) -- 複数の関数を使う
//                   関数参照の追加

#include <iostream.h>

void apple()
{
    cout << "func: apple running..." << endl;
}

int main()
{
    cout << "func: main running..." << endl;
    apple();     // ここで関数appleを実行する
    orange();    // ここで関数orangeを実行する

    return 0;
}

void orange()
{
    cout << "func: orange running..." << endl;
}
```

関数mainに注目されたい．関数参照を使ってそれぞれの関数を呼び出しているのがわかる．

```
int main(int ac, char *av[])
{
    cout << "func: main running..." << endl;
    apple(); ―――――――――――― ここと
    orange(); ――――――――――― ここ
```

```
    return 0;
}
```

2.3◆コンパイルエラー

残念ながら，このプログラムをコンパイルするとエラーになる．

図12-1▼

```
% g++ go.C
go.C: In function `int main(int, char **)':
go.C:15: warning: implicit declaration of function `int orange(...)'
/tmp/cca002291.o: In function `main':
/tmp/cca002291.o(.text+0x5f): undefined reference to `orange'
% _
```

なぜコンパイルエラーになるのだろう？　関数参照の書き方が悪いのだろうか？

このエラーメッセージを見ると，不思議なことに，orangeの名前だけが報告されている．appleの関数参照ではエラーになっていないが，orangeの方はエラーになっている．これはどうしたことだろう？　どちらも同じ書き方をしているのに….

```
apple(); ──────────── これはエラーになっていない
orange(); ─────────── これはエラーになっている
```

以上の疑問については，次の「**関数のプロトタイプ宣言**」のところで明らかになる．

3 関数のプロトタイプ宣言

プロトタイプ（*prototype*）というのは，模型とか試作品といった意味である．**関数のプロトタイプ宣言**は，関数定義の模型である．

3.1◆プロトタイプ宣言の書式

一般的な関数定義は次のような形をしている．

```
void orange() ―――― 関数ヘッダ
{
      ………        ―― 関数の本体
}
```

それに対して，関数のプロトタイプ宣言は，次のような形をしている．

```
void orange()；
```

これは，関数定義から関数の本体を取り去ったもの（これを**関数ヘッダ**という）にセミコロンを付けたものである．

関数のプロトタイプ宣言は，関数定義としては不完全である．関数のヘッダだけで，関数の本体がない．だからプロトタイプである．

3.2◆関数参照のチェック

関数のプロトタイプ宣言には，関数の本体がない．それが，何の役に立つのだろう？　誰がそれを利用するのだろう？

プロトタイプ宣言は，コンパイラによって使われる．関数のプロトタイプ宣言には関数の本体がないので，その関数にどのような文が含まれているかはわからない．しかし，関数の名前や型，そして引数の型はわかる（引数については第13章で説明する）．コンパイラは，これらの情報を使って，関数参照が正しく行われているかをチェックする．

たとえば，次のような関数プロトタイプ宣言があったとする．

```
void orange();
```

これによりコンパイラはorangeという名前が関数であり，その型がvoidであることを知る．それに対して，プログラマーが，誤って次のような関数参照をしたとする．

```
int x = orange();
```

これは，関数orangeが返す値（関数値）をint型の変数xに代入している．この使い方は間違っている．なぜなら，orangeはvoid型なので値を返さない．値がないので代入することはできない．

コンパイラはプロトタイプ宣言によってorangeがvoid型であることを知っている．ゆえに，この使い方が間違っていることがわかるので，プログラマーに対してエラーを報告することができる．実際，このようなプログラムを書くと，次の

ようなコンパイルエラーになる．

図12-2▼

```
% g++ go.C
go.C: In function `int main(int, char **)':
go.C:15: invalid use of void expression
% _
```

このように関数のプロトタイプを宣言することにより，関数の使い方のチェックをすることができる．

3.3◆CとC++のプロトタイプ宣言

古いCには，関数のプロトタイプ宣言がなかった．新しいCなら，関数のプロトタイプ宣言ができる．ただし，Cでは，それが必須ではない．プロトタイプ宣言をすれば，誤った関数の使い方を注意してくれるし，宣言をしなければ注意をしてくれない．

しかし，C++ではmain関数を除くすべての関数について，プロトタイプ宣言が必要である．このため，C++では誤った関数の使い方ができないようになっている．

ここで，第2節のサンプルプログラムを振り返る．このプログラムでは，3つの関数が使われていた．

main
apple
orange

これらの関数のうち，mainを除く残りの関数については，関数のプロトタイプ宣言が必要である．ところが，第2節のサンプルプログラムでは，それが欠けていた．これが，コンパイルエラーになった理由である．

そこで，次のプロトタイプ宣言を追加する．

```
void apple();
void orange();
```

これらのプロトタイプ宣言を，これらの関数が使われる前に置く．そうすれば，コンパイルエラーを取り除くことができる．

しかし，待てよ．あのときのコンパイルでは，関数orangeの方はエラーになったが，関数appleの方はエラーにならなかった――？　まだ，完全には疑問が解けていないようである．この疑問については，関数を並べる順序と関数プロトタイプ宣言の関係を調べれば明らかになる．

3.4◆関数参照とプロトタイプ宣言

コンパイラは，プロトタイプ宣言によって，関数の使い方が正しいかどうかをチェックすることができる．これを可能にするためには，関数を使う前に，その関数のプロトタイプが宣言されていなければならない．

【正しい例】

```
void apple(); ―――――― プロトタイプ宣言
apple(); ――――――――― ここでappleを使う
```

この順序を逆にして，関数を使う後からプロトタイプを宣言してもダメである．

【正しくない例】

```
apple(); ――――――――― ここでappleを使う
void apple(); ―――――― プロトタイプ宣言
```

このように，プロトタイプ宣言と関数参照の順序は重要である．必ずプロトタイプ宣言が先になければならない．

3.5◆プロトタイプ宣言の省略

関数定義と関数参照の順序関係によっては，プロトタイプ宣言を省略できる場合がある．次の例を見てみよう．

```
void apple() ―――――― appleの関数定義
{                    関数定義はプロトタイプ宣言も兼ねる
    .........
}

int main()
{
    .........
```

```
        apple();  ────────── ここでappleを使う
        .........
    }
```

この場合，関数appleを定義してからappleを使っている．このような位置関係にあるときは，appleのプロトタイプ宣言を省略することができる．なぜなら，プロトタイプ宣言は，関数定義のサブセットである．関数定義から本体を取り除いた部分である．したがって，関数定義があれば，コンパイラはその関数の使い方がわかるからである．

このように，関数定義が関数を使う前にある場合は，プロトタイプ宣言を省略することができる．もちろん，その場合でも，宣言してもかまわない．

関数を定義してから使うときはプロトタイプ宣言を省略できる
関数を使ってから定義するときはプロトタイプ宣言が必要

これで，先のサンプルプログラムにおけるコンパイルエラーの意味が明らかになった．funcs.Cでは，mainを除くとapple，orangeの2つの関数が使われていた．その定義と参照は，次の順序で行われていた．

```
    void apple()  ────────── appleの関数定義
    {                         appleのプロトタイプ宣言でもある
        .........
    }

    int main()
    {
        .........
        apple();  ────────── ここでappleを使う
        orange(); ────────── ここでorangeを使う
        .........            まだorangeのプロトタイプ宣言は行われていない
    }

    void orange() ────────── orangeの関数定義
    {
        .........
    }
```

関数appleの方は定義してから使っている．したがって，この場合はプロトタイプ宣言が要らない．一方，関数orangeの方は，使ってから定義をしている．この場合は，プロトタイプ宣言が必要である．これが，関数orangeについてだけコンパイルエラーが報告された理由である．

3.6◆複数の関数を使う（改良版）

次に，第2節のサンプルプログラムfuncs.Cにプロトタイプ宣言を追加し，コンパイルできるようにしたものを示す．

【サンプルプログラム】

```
// funcs.C(第3版) -- 複数の関数を使う
//                   プロトタイプ宣言の追加

#include <iostream.h>

void apple();     // この宣言は省略できる
void orange();    // この宣言は省略できない

void apple()
{
    cout << "func: apple running..." << endl;
}

int main()
{
    cout << "func: main running..." << endl;
    apple();     // ここで関数appleを実行する
    orange();    // ここで関数orangeを実行する

    return 0;
}

void orange()
{
    cout << "func: orange running..." << endl;
}
```

このプログラムは，コンパイルエラーにはならない．次に実行例を示す．

【実行例】

```
% a.out
func: main running...
func: apple running...
func: orange running...
% _
```

3.7◆C方式とPascal方式

第1.7節「**関数を並べる順序**」のところで,「1つのプログラムを複数の関数で構成する場合,関数を並べる順序は自由である」ことを説明した.しかし,プロトタイプ宣言と関数参照との関係が絡んでくると,**関数を並べる順序**にも意味が出てくる.

確かに,関数を並べる順序は**自由**である.しかし,上手に関数を並べると,プロトタイプをまったく宣言することなくコンパイルすることができる.ここにおいて好みの問題が出てくる.

すべての関数についてプロトタイプを宣言するのが好みの人もいれば,関数を並べる順序を工夫し,できるだけプロトタイプ宣言をしないようにするのが好みの人もいる.もちろん,どちらでもかまわない.前者のタイプの人は,関数を次の順序で並べるのが一般的である.

```
int main()
{
    ............
}

void foo()
{
    ............
}

............
```

すなわち,まず関数mainを先頭に置く.続いて関数mainの中で使われている関数を並べ,さらにそれらの関数の中で使われる関数を並べる.関数を並べる順序は,おおむね,それらの関数が実行される順序である.これを**C方式の関数配置**と

いう．初期のCのプログラムに，このような配置がよく見られるからである．

後者の人は，これとちょうど逆の順序になる．

```
............

void foo()
{
      ............
}

int main()
{
      ............
}
```

すなわち，関数mainはいちばん最後に置かれる．続いて関数mainの中で使われている関数をその前に並べ，さらにそれらの関数の中で使われる関数をその前に並べる．関数を並べる順序は，定義してから使うという順序である．これを**Pascal方式の関数配置**という．

Pascalのプログラムは，定義してからでないと名前を参照できないので，自然とこのような配置になるからである．

C方式の関数配置を採用するのも，**Pascal方式の関数配置**を採用するのも自由である．本書では，後者の配置を採用する．すなわち，できるだけプロトタイプ宣言をしないで済むように関数を並べる．

4 なぜ関数を使うのか

少し長い道のりであったが，関数の定義からプロトタイプ宣言までを説明してきた．これで自由に関数を定義して使うことができるようになった．ここでは，

- なぜ関数を使うのか？
- どのようなとき関数を定義するのか？

といった問題を整理しておく．

4.1◆同一処理の省力化

同じプログラムの中で，同じ処理を何回か繰り返すことがある．

```
int main()
{
    …………
    処理A        // 処理Aは複数の文で構成される
    …………     // ここにも複数の文がある
    処理A
    …………
    処理A
    …………
}
```

この例では，**処理A**が3箇所に見られる．**処理A**が1つか数個の文なら，このように書いてもよい．しかし，**処理A**がかなりまとまった文からなるとき，同じ処理を何回も書くのは合理的とはいえない．

幸いにも**処理A**が同じ場所にあれば反復文が使える．しかし，このように繰り返すべき処理が離れたところにある場合は，反復文が使えない．

このようなときは，**処理A**を関数にまとめるとよい．繰り返す場所でその関数を参照するようにすれば，同じ処理を何回も書かなくて済む．

```
void func_a()
{
    // ここに処理Aの内容を書く
}

int main()
{
    …………
    func_a();   // 処理Aを参照する
    …………
```

```
        func_a();    // 処理Aを参照する
        ............
        func_a();    // 処理Aを参照する
        ............
    }
```

4.2◆困難は分割せよ

同じ内容が繰り返されるわけではなくても，まとまった部分を関数化することがある．たとえば，次は，ある長いプログラムをmain 1つの中に閉じ込めた例である．

```
    int main()
    {
        文1
        文2
        文3
        ............
        文10000

    }
```

関数mainには10,000個の文が詰め込まれている．かなり大きなプログラムである．この例は極端であるが，かなりたくさんの文を1つの関数の中に閉じ込めることは可能である．文法的にも間違いではない．しかし，そのようなプログラムは読みにくい．

このような場合，よく観察すると，複数の文が少しずつのまとまりに分かれていることがある．たとえば，いまの例の場合，10,000個の文が，おおまかに3つのパートに分かれるとする．

```
    int main()
    {
        処理A        // 文1～文3000
        処理B        // 文3001～文7000
        処理C        // 文7001～文10000
    }
```

このような場合は，それぞれのパートを1つの関数にまとめるのである．

```
void func_a()
{
    // ここに処理Aの内容を書く
}

void func_b()
{
    // ここに処理Bの内容を書く
}

void func_c()
{
    // ここに処理Cの内容を書く
}

int main()
{
    func_a();   // 処理Aを参照する
    func_b();   // 処理Bを参照する
    func_c();   // 処理Cを参照する
}
```

この結果，10,000個の文で構成されていたmainが，たった3個の文に縮小される．mainだけに限れば，かなり読みやすいプログラムになった．

しかし，func_a，func_b，func_cには，まだそれぞれ数千個の文が含まれている．そこで，それぞれの関数も，さらに細分化する．たとえば，func_aを，さらに5つに分解するといったように….

```
void func_a_1()
{
    // ここに処理Aのパート1を書く
}

..................................
```

```
void func_a_5()
{
    // ここに処理Aのパート5を書く
}

void func_a()
{
    func_a_1();            // 処理Aのパート1を実行する
    func_a_2();            // 処理Aのパート2を実行する
    func_a_3();            // 処理Aのパート3を実行する
    func_a_4();            // 処理Aのパート4を実行する
    func_a_5();            // 処理Aのパート5を実行する
}
```

このように大きな関数を小さな関数に分解していくことにより，やがて1つ1つの関数が読みやすい単位の大きさに収束していく．

一般的に，1つの関数は，1つの処理だけに専念すべきである．もし複数の処理が含まれているようなら，その関数を分解できないか検討すべきである．

関数を小さくすることにより，プログラムを読みやすくすることができる．また，プログラムの誤り（バグ）を見つけやすく，修正しやすくすることができる．誤りを修正しても，その修正を関数の中にとどめることができる．修正が他の関数に影響を与えることはない．

目安は，1つの関数が1画面に収まるくらいに小さくすることである．ただし，switch文で多重分岐するときなどは，複数の画面にまたがることもある．

4.3◆プログラムの部品化

関数を「プログラムを**部品化**するための手段」として利用することがある．たとえば次は，「**四則演算子**」のところで取り上げた1〜1000までの合計を表示するプログラムである．

```
int main()
{
    cout << (1 + 1000) * 1000 / 2 << endl;
    return 0;
}
```

非常に短いプログラムなので，この中をさらに関数化する必要はなさそうである．ここで，関数mainの最初の文に注目する．

```
cout << (1 + 1000) * 1000 / 2 << endl;
```

これは，1〜1000までの合計を求め，さらにその結果を出力するというものである．この中の「1〜1000までの合計を求める」部分に注目する．

```
cout << (1 + 1000) * 1000 / 2 << endl;
```

ここ

これは，1〜1000までの合計を求めるというプログラムの1部品である．この部分を関数化しておけば，別のプログラムで再利用できるかもしれない．そこで，次のような関数を考える．

```
int sum()
{
    return (1 + 1000) * 1000 / 2;
}
```

sumは，1〜1000までの合計を求めるための関数である．結果を返すので関数の型はvoidではない．intである．この関数を使うと，もとのプログラムは次のように書き換えられる．

【サンプルプログラム】

```
// count1000.C(第2版) -- 1〜1000までの合計を表示する

#include <iostream.h>

int sum()
{
    return (1 + 1000) * 1000 / 2;
}

int main()
{
    cout << sum() << endl;
    return 0;
}
```

関数sumを呼び出す

【実行例】

```
% a.out
500500 ─────────── 1〜1000までの合計が計算され，
                    その結果が表示される
% _
```

これをもとのプログラムと比べると，関数化した分だけ太ってしまった．しかし，関数sumを得たことにより，プログラム再利用の道が開けた．もし別のプログラムで1〜1000までの合計が必要になったら，この関数を利用することができる．

注意　関数sumは1〜1000までの合計に制限される．任意の数から任意の数までの合計を求めることはできない．しかし「引数」を導入すると，この制限を解除することができる．
関数の引数については，第13章で説明する．

このように，関数は，プログラムを部品化するための手段として利用される．関数の使い方としては，この使い方が最もポピュラーで，また有効である．
プログラムを作りながら，プログラムの部品として関数を蓄積していく．やがて，それが大きな財産になることだろう．

5 関数値

この節では，関数値にまつわる事項をまとめる．

5.1◆値を返す関数と返さない関数

関数には，値を返すものと返さないものがある．

```
void subroutine(); ──── この関数は値を返さない
double amount(); ──── この関数はdouble型の値を返す
```

関数が返す値を**関数値**という．void型以外の関数は値を返す．

関数が値を返すとき，その関数は値を必要とする場面で参照される．たとえば，

変数に代入したり，coutの中で使われたりのように．

```
sum = amount();        ――― 関数の値が変数に代入される
cout << amount();      ――― 関数の値を表示する
```

値を返さない関数（void型の関数）を，値を必要とする場面で参照することはできない．つまり，次はいずれもコンパイルエラーになる．

```
sum = subroutine();    ――― void型関数を変数に代入しようとした
cout << subroutine();  ――― void型関数の値を表示しようとした
```

逆に，値を返す関数を，値を必要としない場面で使うこともできる．

```
amount();              ――― 関数としての処理は行われるが,
                           その関数値は捨てられる
```

5.2◆べき乗 pow

第5.4節で「10年後の元利合計」を求めるプログラムを作る予定であるが，その中で使用するpowという関数を説明する．

次は，第4章第2節で示した算術演算子である．

演算子	演算
+	加算
-	減算
*	乗算
/	除算
%	剰余

剰余を含めて，ひと通りの演算子が揃っている．しかし，面白い（困った）ことに，C++にはべき乗を行う演算子が存在しない．もし，べき乗を必要とするなら，自分でそのためのプログラムを書かなければならない．

それではあんまりなので，powという関数が用意されている．powの書式は，次のようになっている．

```
double pow(double x, double y);
```

たとえば，10の6乗を計算したければ次のように書く．

```
pow(10, 6);
```

powの返す値は，doubleである．関数powを使うときは，math.hをincludeする必要がある．

```
#include <math.h>
```

5.3◆static変数（関数内部）

第5.4節で作成予定の「10年後の元利合計」を求めるプログラムについて，もう1つ説明しておきたい事項がある．それは，関数内部で宣言される**static変数**についてである．

●関数内部の変数

一般に，関数内部の変数は，次のように宣言される．

```
int func()
{
    int var = 100; ―――――――― (1)
    return var++;
}
```

ここで宣言されている変数varに着目する．その初期値は，100である．この関数は，そのvarの値を関数値にしている．したがって，この関数を実行すると，100が返される．

```
cout << func(); ―――――― 100が表示される
```

ところで，varは++演算子で+1されている．したがって，varの最終的な値は101である．しかし，この値は意味を持たない．なぜなら，次にこの関数が実行されたとき，(1) の文によって再び値が100に戻されるからである．つまり，この関数を何回実行しても100が返される．

```
第1回目の実行: func()の値= 100
第2回目の実行: func()の値= 100
第3回目の実行: func()の値= 100
```

```
第4回目の実行: func()の値= 100 ――――― 何回実行しても値は100
…………………………………………
```

●static変数

ところで，(1) の宣言でstaticを付けると，話は変わってくる．

```
static int var = 100;
↑
ここ
```

このようにstatic宣言された変数を**static変数**という．この場合，varはstatic変数になる．

static変数が普通の変数と異なるのは，初期化が1回しか行われないということである．たとえば，次の (2) でstatic変数varを100に初期化している．

```
int func()
{
    static int var = 100; ―――――――― (2)
    return var++;
}
```

しかし，この初期化は1回しか行われない．2回目以後にこの関数が実行されたときは，(2) の文は無視される．そして，前回のvarの値がそのまま有効になる．したがって，funcを実行するたびにvarの値は1つずつ大きくなっていく．

```
第1回目の実行: func()の値= 100
第2回目の実行: func()の値= 101
第3回目の実行: func()の値= 102
第4回目の実行: func()の値= 103
…………………………………………
```

この点が，一般の変数とstatic変数の違いである．

5.4◆10年後の元利合計

1年間の利率が3.5%のとき，10万円を10年間預けると元利合計（元金と利息の合計）はいくらになるかを求めるプログラムを作成する．

この節では，関数値について学んでいる．そこで，元利合計を求める部分を関数化して，その関数の関数値が元利合計になるようにする．すなわち，関数名を

amountとすれば，次のようにamountを実行するたびに1年後の元利合計を返すようにする．

```
amount();  ―――― 0年後の元利合計
amount();  ―――― 1年後の元利合計
amount();  ―――― 2年後の元利合計
…………
amount();  ―――― 10年後の元利合計
```

この仕様を見ると，amountは毎回返す値が異なる．これは，第5.3節で説明したstatic変数を使えば実現できる．

```
double amount() ―――― amountは元利合計をdouble型で返す
{
    static int year = 0; ―― yearは何年後かを示す
    …………
}
```

一般に，元利合計は次の式で計算できる．

元金 ×（1＋利率）^{年数}

すなわち，べき乗の計算が必要である．C++にはべき乗を計算する演算子がないので，第5.2節で説明したpowを使う．したがって，amountは，次のようになる．

```
double amount()
{
    static int year = 0;

    return 100000.0 * pow((1 + 0.035), year++);
}
```

関数値として元利合計を返しながら，同時にyearを+1していることに注意．

main関数で，このamountを参照する．せっかくだから，0年後（すなわち，預けた直後）から10年後までの元利合計を毎年表示するようにする．

```
for (int i = 0; i <= 10; i++)
    cout << setw(2) << i << "年後の元利合計 = "
         << amount() << "円" << endl;
```

年を表示するとき，1桁の年と2桁の年があるので，桁がずれないようにsetw(2)を

使っている．したがって，このプログラムでは次の2つのincludeが必要である．

```
#include <iomanip.h> ――― これはsetw()を使うため
#include <math.h> ――――― これはpow()を使うため
```

以上をまとめると，次のようなプログラムが出来上がる．

【サンプルプログラム】

これは，元金10万円で1年間の利率が3.5%のときの元利合計を10年目まで表示するものである．

```
#include <iomanip.h>        // for setw()
#include <math.h>           // for pow()

double amount()
{
    static int year = 0;

    return 100000.0 * pow((1 + 0.035), year++);
}

int main()
{
    for (int i = 0; i <= 10; i++)
        cout << setw(2) << i << "年後の元利合計 = "
                 << amount() << "円" << endl;

    return 0;
}
```

【実行例】

```
% a.out
 0年後の元利合計 = 100000円
 1年後の元利合計 = 103500円
 2年後の元利合計 = 107122円
 3年後の元利合計 = 110872円
 4年後の元利合計 = 114752円
 5年後の元利合計 = 118769円
 6年後の元利合計 = 122926円
 7年後の元利合計 = 127228円
 8年後の元利合計 = 131681円
 9年後の元利合計 = 136290円
10年後の元利合計 = 141060円
% _
```

6 インライン関数

この節では，関数のコードを直接呼び出し位置に展開する**インライン関数**について説明する．

6.1◆inline

いま，次のような関数があるとする．

```
type func()
{
    ............
}
```

ここでtypeは，voidを含む任意の型である．このとき，typeの前に**inline**というキーワードを付けたものを**インライン関数**という．

```
ここ
↓
inline type func() ←―― funcはインライン関数になる
{
    ............
}
```

関数をインラインにしても，使用上の違いは生じない．普通にその関数を呼び出して使うことができる．

6.2◆インライン展開

一般の関数とインライン関数とどこが違うのかというと，実行ファイルの作られ方が違う．コンパイルのされ方が違う．インライン関数のコードは，関数の呼び出し位置に関数本体のコードが直接埋め込まれる．これを**インライン展開**という．

たとえば，次のようなインライン関数とその関数呼び出しがあるとする．

```
inline void foo() ―――― これはインライン関数
```

```
{
    文A ―――――― このインライン関数は
    文B          2つの文で構成される
}

int main()
{
    文1
    foo(); ―――――― インライン関数の呼び出し
    文2
    文3
    foo(); ―――――― インライン関数の呼び出し

    return 0;
}
```

このプログラムをコンパイルすると，あたかも次のプログラムをコンパイルしたかのように実行ファイルが作られる．

```
int main()
{
    文1
    文A ◀―――――― インライン関数のコードが
    文B ◀―――――― 展開される
    文2
    文3
    文A ◀―――――― インライン関数のコードが
    文B ◀―――――― 展開される

    return 0;
}
```

インライン関数のコードが直接呼び出し位置に展開されていることに注意．このため，インライン関数を使うと関数呼び出し時におけるオーバーヘッドを最小にできる．

一般に，関数を呼び出すと，そのコードを直接実行するときよりも余計に時間がかかる．また，余計なコードも必要になる．引数の引き渡しや関数値を受け取

るためのコードや，もとの位置に戻るための処理が必要になるからである.

6.3◆メリットとデメリット

インライン関数を使うと関数呼び出し時におけるオーバーヘッドを最小にできるので，実行時間を速くすることができる．しかし，そのインライン関数を何度も使う場合は，実行ファイルのサイズが大きくなってしまう．インライン関数のコードが何回も展開されるからである．

このため，次のような関数がインライン関数に向く．

- 1～3行程度の小さい関数
- 引数が多く，複雑な関数
  （関数呼び出しのオーバーヘッドが大きい）
- 呼び出す回数が少ない関数

インライン関数については，次のような制限がある．

- 関数内に反復文がある場合はインライン関数にできない
- 使用する前に定義されていなければならない
  inline付きのプロトタイプ宣言はできない
- 有効範囲は，定義からファイルの終わりまで
  別ファイルで定義されているインライン関数を使うことはできない

インライン関数は，Cのマクロに相当するものである．ただし，次の点で，Cのマクロよりも有効である．

- 引数の展開が関数と同様に行われる
  このため，マクロによる副作用がない
- 引数や戻り値の型のチェックが行われる

第13章 引数

これまで，たくさんの関数を書いてきた．しかし，引数については使ってこなかった．関数を定義するとき，引数を指定する（）の中を空っぽの状態で定義してきた．

この章では，関数の引数について説明する．実際に引数を定義して，それを使用する．また，引数にまつわるさまざまな事項についても説明する．

1 引数

第12章第4.3節「プログラムの部品化」のところでsumという1～1000までの合計を求める関数を紹介した．

```
int sum()
{
    return (1 + 1000) * 1000 / 2;
}
```

この関数をプログラムの部品として見ると，あまり使い勝手がよくない．なにせ合計できる範囲が1～1000に限定されている．これを任意の範囲で合計を計算できるように汎用化すると，ずっと使い途が増える．そのためには，使い方によってsumの挙動を変える必要がある．

あるときは1～100までの合計を求める
あるときは1～10000までの合計を求める
あるときは200～400までの合計を求める

このようなことを可能にするには，関数を使うときに合計する範囲を指定できるようにする必要がある．それを可能にしてくれるのが，関数の**引数**である．

1.1◆仮引数の定義

関数を参照するとき，関数に対して引数を渡すことができる．そのためには，関数を定義するとき，いっしょに**仮引数**を定義する必要がある．

仮引数というのは，名前のとおり，仮の引数である．本当の引数は，その関数を使うときに指定される．その本物の引数の値が仮引数に代入されるので，関数定義の中では仮引数を普通の変数のように使うことができる．

仮引数は，関数名の後の（）の中で定義する．

ここ

```
型 関数名(仮引数の定義)
{
    文
    ……
}
```

仮引数の定義は，次のように仮引数の型と名前を組にして指定する．

```
型 仮引数名
```

次に例を示す．

```
void func(int nval)
```

この例では，nvalというint型の仮引数を定義している．
複数の仮引数が必要な場合は，仮引数をカンマで区切って並べる．

```
型 仮引数名, 型 仮引数名,……
```

たとえば次の関数funcは，int型の仮引数nとfloat型の仮引数valを定義している．

```
void func(int n, float val)
```

1.2◆引数付き関数の参照

引数が定義されている関数を実行するときは，次のように，与える引数をカンマで区切って並べる．

関数名(引数1, 引数2, ……)

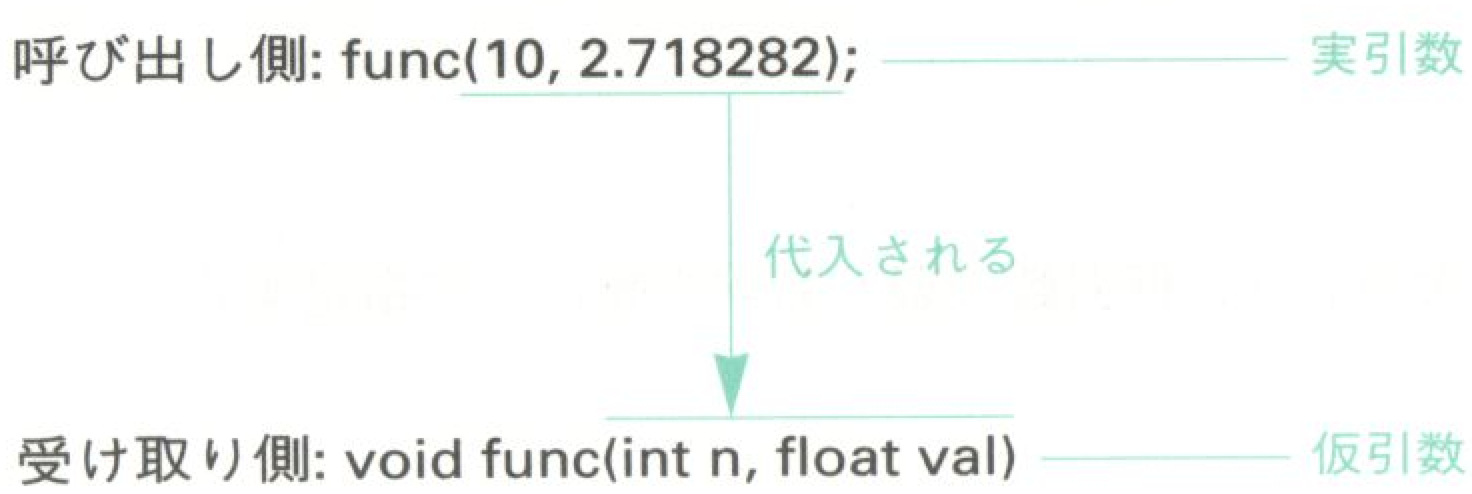
たとえば，先の関数funcを呼び出すときは，次のようにする．

```
func(10, 2.718282);
```

ここで指定した引数10や2.718282を仮引数に対して**実引数**という．指定された実引数は，関数funcの仮引数に代入される．

```
n = 10
val = 2.718282
```

つまり，関数を呼び出す側の第1引数10が関数定義側の第1仮引数nに代入され，第2引数の2.718282が定義側の第2仮引数valに代入される．



1.3◆min～maxまでの合計

引数付き関数の例として，1～1000までの合計を求める関数を汎用化する．引数を使って「いくつからいくつまでの合計を求める」のかを指定できるようにする．小さい方の数をmin，大きい方の数をmaxとすれば，関数の頭書きは次のようになる．

```
int sum(int min, int max)
```

minからmaxまでの整数の合計は次の式で計算することができる（簡単な算数）．

```
  min + (min + 1) + (min + 2) +……+ max
= (min + max) * (max - min + 1) / 2
```

したがって，関数sumは次のようになる．

```
int sum(int min, int max)
{
    return (min + max) * max / 2;
}
```

この関数の利用例をいくつか示す.

```
x = sum(1, 10);
        ──────── 1～10までの合計を変数xに代入する
cout << sum(200, 300) << endl;
        ──────── 200～300までの合計を表示する
```

【サンプルプログラム】

これは，関数sumの使用例を示すものである．関数sumを使って200～300までの合計を表示している．

```
// 200～300までの合計を表示する

#include <iostream.h>

int sum(int min, int max)
{
    return (min + max) * max / 2;
}

int main()
{
    cout << sum(200, 300) << endl;
    return 0;
}
```

【実行例】

```
% a.out
75000 ──────────── 200～300までの合計が表示される
% _
```

2 デフォルト引数

いま，funcという関数があって，整数の引数を1つとるとする．

```
func(1000);
```

この関数に与える引数は，いつも1000であることが多いとする．もし100%そうならば引数なしにすることもできるが，しかし，ごく稀(まれ)に他の値を必要とすることもあるので，引数を削ることもできない．でも，やはり毎回1000を指定するのは面倒….

このようなときは，**デフォルト引数**を使うことができる．

2.1◆デフォルト引数

デフォルト引数というのは，関数を参照するときに引数の指定を省略すると，あらかじめ決められた値が指定されたものと見なしてくれる引数である．

```
func();          ―― 引数を指定しない
                    この場合、1000が指定されたものと
                    見なしてくれる

func(1024)       ―― 同じ関数で引数を指定することもできる
```

デフォルト引数を使うには，関数定義で仮引数に代入を使って初期値を指定すればよい．

```
void func(int arg)          ―― これは普通の引数
{
    ............
}

void func(int arg = 1000)   ―― これはデフォルト引数
{                              実引数が省略されると1000になる
    ............
}
```

2.2◆複数の引数がある場合

引数が複数個あって，そのうちの一部をデフォルト引数にしたいときは，必ず最後の引数から始めなければならない．

たとえば，x，y，zという3つの引数があって，そのうちの2つをデフォルト引数にしたいとする．この場合，次のような並びはダメである．

```
void func(int x = 200, int y, int z =300)
```

デフォルト引数の後ろに普通の引数がある

途中に普通の引数が混じっているからである．デフォルト引数にしたい引数は，必ず最後の方に並べなければならない．たとえば，次のように．

```
void func(int x, int y = 200, int z = 300)
```

デフォルト引数は最後に並べる

2.3◆サンプルプログラム

次に，デフォルト引数を使ったサンプルプログラムを示す．

【サンプルプログラム】

このプログラムに使われているfuncには，デフォルト引数が使われている．デフォルト引数の位置に注意．

```
#include <iostream.h>

void func(int x, int y = 200, int z = 300)
{
    cout << "第1引数 = " << x << endl;
    cout << "第2引数 = " << y << endl;
    cout << "第3引数 = " << z << endl;
}

main()
```

```
{
    func(100); ――――――――――――――― 実引数が1つ
}

% a.out
第1引数 = 100
第2引数 = 200
第3引数 = 300
% _
```

3 関数の多重定義

いま，2数を与えて，大きい方の値を返す関数maxを作りたいとする．

max(100, 200) ―――――― 関数の値は200になる

maxをできるだけ汎用的な関数に仕上げたいところではあるが，少し問題がある．それは，ひと口に「数」といっても，整数と浮動小数点数があり，それぞれ別のデータ型だからである．このため，整数版のmaxと浮動小数点数版のmaxとで関数を分けなければならない．

```
int intmax(int x, int y); ―――――― 整数版のmax
float floatmax(float x, float y); ―― 浮動小数点版のmax
```

ここで，整数版maxと浮動小数点版maxとで関数名を変えてあることに注意．C言語の場合は，たとえ処理内容が同じでも，扱う型が異なれば異なる名前の関数を用意しなければならなかった．

【例】 atoi ―――――――――― 文字列を整数に変換する
atof ―――――――――― 文字列を浮動小数点に変換する

C++は，同じ名前で複数の関数を定義することができる．これを**関数の多重定義**または**関数のオーバーロード**（*overload*）という．

3.1◆多重定義の条件

C++では，関数の形が異なれば，同じ名前の関数を複数個定義できる．ここで，

「形が異なる」というのは，引数が異なることを意味する．関数値の型が違ってもダメである．

引数の型が異なるというのは，次のいずれかの場合である．

- **引数の型が異なる**
- **引数の数が異なる**

たとえば，次の2つの関数toddは引数の型が異なるので多重定義可能である．

```
void todd(int x);
void todd(char c);
```

また，次の2つの関数kellyは引数の数が異なるので多重定義可能である．

```
void kelly(int a);
void kelly(int a, int b);
```

しかし，次の2つの関数nicolは関数値の型は異なるが，引数が同じなので多重定義することはできない．

```
int kelly(int a, int b);
char kelly(int a, int b);
```

次の2つの関数johnは，引数の名前が異なるので多重定義できるように見える．しかし，引数の型が同じなので多重定義することはできない．

```
void john(int a);
void john(int x);
```

3.2◆max & min

それでは，この節の導入のところで提起した関数maxを作成する．まず，次が整数版のmaxである．

```
int max(int x, int y)
{
    if (x > y)
        return x;
    else
```

```
        return y;
}
```

第9章で説明した三項演算子を使うと，もっと簡単に書くこともできる.

```
int max(int x, int y)
{
    return (x > y) ? x : y;
}
```

同様にして，float版のmaxを作成することができる.

```
float max(float x, float y)
{
    return (x > y) ? x : y;
}
```

以上をテストするためのサンプルプログラムを次に示す.

【サンプルプログラム】

これは，2つの数のうち大きい方の値を返すmaxの作成例である．int版とfloat版を多重定義している．また，逆のminも作成している．

```
#include <iostream.h>

int max(int x, int y)            int版のmax
{
    return (x > y) ? x : y;
}

float max(float x, float y)      float版のmax
{
    return (x > y) ? x : y;
}

int min(int x, int y)            int版のmin
{
    return (x < y) ? x : y;
}

float min(float x, float y)      float版のmin
{
```

```
    return (x < y) ? x : y;
}

int main()
{
    cout << max(100, 200) << endl
         << max(3.14f, 2.7f) << endl ―――――― 最後の“f”を忘れないこと
         << min(100, 200) << endl
         << min(3.14f, 2.7f) << endl; ――――― 最後の“f”を忘れないこと

    return 0;
}
```

【実行例】

```
% a.out
200
3.14
100
2.7
% _
```

1つ注意しておくと，浮動小数点数版のmaxやminの呼び出しにおいて，3.14fのようにリテラルの最後に“f”を付けるのを忘れないように！　第4章第3.3節で説明したように，浮動小数点数リテラルは，デフォルトではdoubleになる．関数の多重定義では，floatであることを明確にするため，最後に“f”を付ける．

第14章 スコープ

この章では，名前の**スコープ**について説明する．スコープは，その名前を参照できる範囲を示す．C++は，スコープをブロック，関数，ファイル単位に細かく制御することができる．

1 関数呼び出しを数える

いま，ある関数を呼び出した回数を数えてみたいとする．呼び出す関数の名前をfuncとする．たとえば，次の例ならfuncを3回呼び出している．

```
int main()
{
    func();  ―――― funcを3回呼び出す
    func();
    func();

    return 0;
}
```

これをプログラムを使って数えるとしたらどうするか？

適当な変数（たとえばcount）を用意し，0で初期化する．0で初期化するのは，まだ，funcを呼び出していないからである．そして，funcを呼び出すたびにcountの値を+1していく．こうすれば，変数countにfuncを呼び出した回数が得られる．

```
int main()
{
    int count = 0; ———— 変数countを0で初期化する

    func(); ———— funcを呼び出すたびに
    count++; ———— countを+1する

    func(); ———— 2回目の呼び出し
    count++;

    func(); ———— 3回目の呼び出し
    count++;

    return 0;
}
```

最後にcountの値を出力する．そうすれば，funcを呼び出した回数が出力されるはずである．

```
cout << "func()の呼び出し回数= " << count << endl;
```

この例の場合なら3回呼び出しているので，次のように表示されるはずである．

```
func()の呼び出し回数= 3
```

本当だろうか？　何か落とし穴はないだろうか？

2 ローカル変数

この節では，スコープのうちブロックスコープについて説明する．**ブロックスコープ**を持つ変数を**ローカル変数**という．

2.1◆名前のスコープ

第1節における第1の危惧は，関数funcの中でcountの値を変更していたらどうかというものである．

```
void func()
{
    count = 100;    ———— countの値を100に変更する
}
```

この例では，関数funcの中でcountの値を100に変更している．このため，せっかく関数mainで数えていたcountの値が無駄になってしまうのではないだろうか？

```
int main()
{
    …………
    func();     ———— count = 100になる
    count++;    ———— count = 101になる
    …………
}
```

この危惧については心配いらない．関数mainと関数funcではスコープが違うからである．

スコープ（*scope*）というのは，宣言した名前が使える範囲のことである．変数名も名前なので，スコープがある．

関数の中で宣言された名前のスコープは，その関数の中だけである．他の関数からその名前を使うことはできない．変数countは，関数mainで宣言されたものである．

```
int main()
{
    int count = 0;    ———— 変数countの宣言
    …………
}
```

したがって，変数countのスコープは，関数mainの中に限定される．関数funcの中から変数countを使うことはできない．

```
void func()
{
    count = 100;    ———— この文はエラーになる
}
```

もし，このような無茶なコードを書くと，次のようなコンパイルエラーになる．

```
In function `void func()':
`count' undeclared (first use this function)
```

2.2◆ローカル変数

変数にはスコープがある．このため，関数funcの中から変数countを使うことはできない．では，もしfuncの中でcountという変数を宣言したらどうか？　こうすれば，関数funcの中でcountを使うこともできるし，countの値を変更することもできる．

```
void func()
{
    int count = 100; ——— countの値を100に変更する
}
```

この結果，せっかく関数mainで数えていたcountの値が無駄になってしまうのではないだろうか？　これが第2の危惧である．

```
int main()
{
    …………
    func(); ——————— count = 100になる
    count++; —————— count = 101になる
    …………
}
```

本当かどうかは確かめてみればよい．

【サンプルプログラム】

このプログラムは，関数の内外で同じ変数が使われているとき，その影響を調べるものである．

```
#include <iostream.h>

void func()
{
    int count = 100; ——————————— countの値を100に変更する
```

```
}

int main()
{
    int count = 0; ───────────── countを0に初期化

    func(); ───────────────── count = 100に変更
    count++;; ──────────────── count = 101になる?

    func();
    count++;

    func();
    count++;

    cout << "func()の呼び出し回数 = " << count << endl;

    return 0;
}
```

プログラムの実行結果は次のようになる.

【実行例】

```
% a.out
func()の呼び出し回数 = 3
% _
```

この結果を見ると,関数funcの呼び出し回数が正しく数えられているのがわかる.関数funcの中で行ったcountの変更がmainに影響を与えていない.——すなわち,関数とその関数の呼び出し側で同じ名前を使っても,それぞれの変数がお互いに影響を及ぼしていないのがわかる.

関数の中で宣言された変数を関数内部の**ローカル変数**という.関数内部のローカル変数のスコープは,その関数の内部に限定される.関数mainで宣言されたローカル変数countと,関数funcで宣言されたローカル変数countのスコープは異なる.たとえ名前は同じでも,お互いに影響を及ぼし合うことはなく,それぞれ独立した変数と見なされる.

2.3◆名前の衝突

関数の呼び出し側と関数内部で同じ名前（たとえば変数名）を使うことを**名前の衝突**という．名前の衝突について，次の点は重要である．

C++は名前の衝突を無視できる

関数を使用する側は，その関数内でどのような変数が使われているかを知る必要がない．関数内部の作りに関知する必要がない．その関数が何を処理してくれるのかだけを知っていればよい．

このことは，プログラムの部品としての関数を作りやすくしてくれる．

なぜこのことを強調するかというと，そうでない仕様の言語が存在するからである（たとえばBASIC）．そのような貧弱な仕様の言語では，関数（言語によっては「サブルーチン」と呼ばれる）を作るとき，つねに名前の衝突に気を使わなければならないからである．

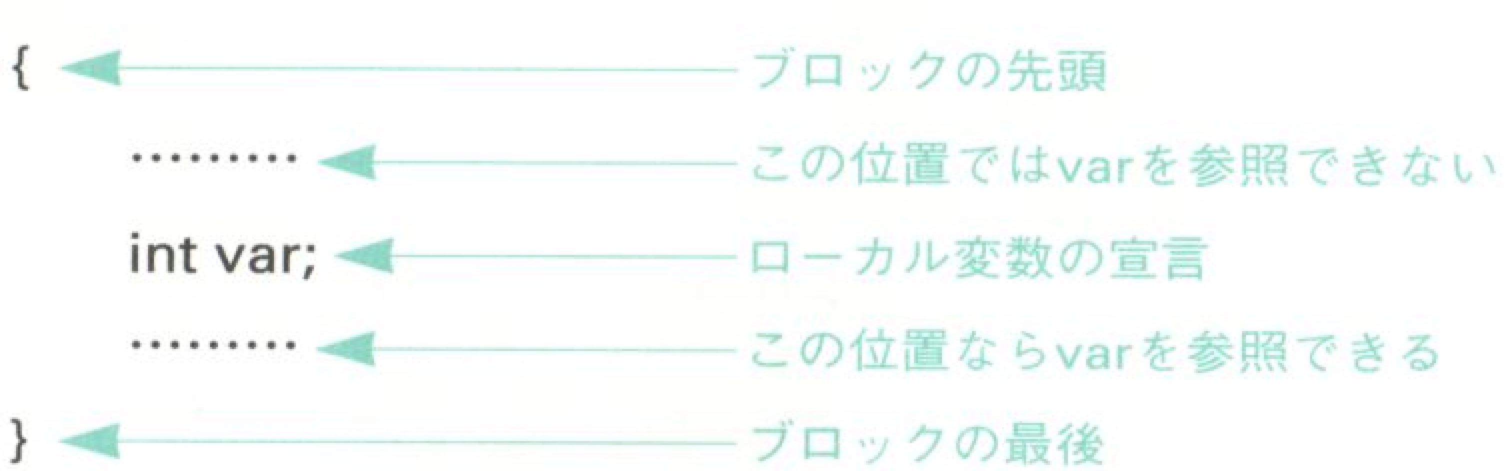
C++では名前の衝突に気を使う必要がない．このため名前に煩わされることなく，気楽に関数を作ることができるのである．

2.4◆ブロックローカル

「**ローカル変数**」のところで，関数の中で宣言された変数を関数内部の**ローカル変数**ということを説明した．もう少し細かくいうと，**ブロック**の中で宣言された変数がローカル変数である．

ブロックというと，複数の文を{}で囲んで1つの文として扱えるようにしたものである．関数の本体も{}で囲まれているので，1つのブロックである．

ブロック内で宣言された変数のスコープは，その宣言位置からブロックの最後までである．



関数の中にブロックを作ることができるので，関数全体よりもさらに小さなスコープを持つローカル変数を宣言することができる．たとえば次のvarは，たとえ同じ

関数内にありながらも，(A) および (B) の位置からは参照することができない．

```
void func()
{
    ............ ←――――――――――― (A)
    {
    int var;
    ............
    }
    ............ ←――――――――――― (B)
}
```

【サンプルプログラム】

このプログラムは，関数の内部にさらに局所的なブロックを作り，その中でローカル変数を宣言している．

```
#include <iostream.h>

int main()
{
    int x = 100;  ―――――― このxはmain全体のスコープを持つ
    cout << x << endl;

    {
        int x = 200;  ―――――― このxはブロック内部にスコープを持つ
        cout << x << endl;
    }

    cout << x << endl;

    return 0;
}

% a.out
100 ―――――― これは関数レベルで宣言されたxの値
200 ―――――― これはブロック内で宣言されたxの値
100 ―――――― 関数レベルxの値が変わっていない
% _
```

2.5◆引数とローカル変数

仮引数は，関数の最も外側のブロックで宣言されたローカル変数と見なされる．

```
void func(int x)
{ ──────────── 最も外側のブロック
}
```

したがって，上の仮引数xは次のように宣言されたローカル変数と同じスコープを持つ．

```
void func()
{
    int x;
    ……
}
```

仮引数はその関数のどこからでも参照することができるが，関数の外からは参照できない．また，同じ理由で仮引数と同じ名前のローカル変数を宣言することはできない．

```
void func(int x)
{
    int x; ──────── 仮引数と同じ名前のローカル変数
}
```

上のようなコードを書くと，次のようなコンパイルエラーになる．

```
In function `void func(int)':
declaration of `x' shadows a parameter
```

3 グローバル変数

時には2つの関数，あるいはそれ以上の関数で同じ変数を共通に使いたい場合がある．しかし，ローカル変数を使うとブロックローカルになるので，ある関数で宣言された変数は，他の関数からは参照できない．

```
void func_a()
{
      int var_a; ──────── var_aはfunc_bからは参照できない
      ............
}

void func_b()
{
      int var_b; ──────── var_bはfunc_aからは参照できない
      ............
}
```

そのようなときは，**グローバル変数**を使う．

3.1◆ファイルスコープ

グローバル変数とは，すべてのブロックの外で宣言された変数である．具体的には，関数，および第2部で説明するクラスの外で宣言された変数である．グローバル変数を使えば，複数の関数から参照できる．

```
int gvar; ──────────── 関数の外で宣言されているのでグローバル変数

void func_a()
{
      gvar = 1; ──────── func_aで設定したgvarの値を
}

void func_b()
{
      if (fvar == 1) ──────── func_bで参照することができる
      ............
}
```

グローバル変数のスコープは，ファイル単位になる．これを**ファイルスコープ**という．すなわち，宣言された位置からファイルの終わりまでの範囲にあるすべての関数から参照することができる．

3.2◆ローカル変数との衝突

いま，グローバル変数とローカル変数で同じ名前を付けたとする．

```
int var = 100; ――――― グローバル変数

void func()
{
    int var = 10; ――――― 同じ名前のローカル変数

}
```

このとき関数funcの中でvarを参照すると，グローバル変数とローカル変数のどちらの値が読み出されるだろう？　もしグローバル変数の値が参照されるならvarの値は100になる．逆に，ローカル変数の値が参照されるならvarの値は10になる．

どちらの結果になるかを，次のサンプルプログラムでテストする．

【サンプルプログラム】

```
#include <iostream.h>

int var = 100;

int main()
{
    int var = 10;
    cout << "var = " << var << endl;
    return 0;
}
```

【実行例】

```
% a.out
var = 10 ――――― これはローカル変数の値
% _
```

この結果を見ると，ローカル変数の値が参照されているのがわかる．このように同じ名前の変数がグローバル変数とローカル変数で宣言されていると，ローカル変数が参照される．

3.3◆スコープ演算子

グローバル変数とローカル変数で同じ名前を付けると，ローカル変数が参照される．しかし，時にはグローバル変数の値を参照したいことがある．そのようなときは，変数名の前に :: (2つのコロン) を付ければよい．

::var ――――― これはvarがグローバル変数であることを示す

:: を**スコープ演算子** (*scope operator*) という．スコープ演算子は，二項演算子である．

【サンプルプログラム】

次のプログラムは，グローバル変数とローカル変数で同じ名前の変数が宣言されている．その両方の変数の値を表示する方法を示している．

```
#include <iostream.h>

int var = 100;

int main()
{
    int var = 10;

    cout << "グローバル変数の var = " << ::var << endl;
    cout << "ローカル変数の   var = " << var << endl;

    return 0;
}
```

【実行例】

```
% a.out
グローバル変数の var = 100
ローカル変数の   var = 10
% _
```

第15章 配列

配列は変数の一種である．1つの名前で複数のデータを記憶することができる．個々のデータは，番号（インデックス）で識別することができる．配列に記憶するデータの種類は，同じ型のデータに限定される．複数の同じ型のデータを1つの名前で管理したいときは，配列が適している．

この章では，配列の基本について説明する．

1 基本型とユーザー定義型

C++に最初から組み込まれているデータ型を**基本型**という．基本型については，第3章の「いろいろなデータ型」のところで説明した．C++で使われる代表的な基本型を次に示す．

データ型	データの種類
文字	1バイトの1文字を表す 　漢字は2バイトなので文字型では表現できない
文字列	文字の並び 　0文字でも，1文字でも，2文字以上でもよい 　漢字を含めることもできる
整数	小数点を含まない数 　8進数や16進数で表現することもできる
浮動小数点数	小数点を含む数

必要ならば，基本型を発展させて，より複雑な型を作ることもできる．これを**ユーザー定義型**という．配列は，ユーザー定義型の1つである．

基本型	：言語に最初から組み込まれているデータ型
ユーザー定義型	：ユーザーが定義したデータ型

2 使い方の基本

配列は，ユーザー定義型の1つであり，指定したデータ型の集合である．1つの変数で複数のデータを扱うことができる．かつ，個々のデータは番号（インデックス）で識別できる．いわばランダムアクセスのメモリ版といえる．

この節では，配列を宣言し，それを使うまでの流れをひと通り説明する．

2.1◆宣言とサイズ

配列を使用するには，変数と同じように**宣言**が必要である．

```
型　配列名[定数式]；
```

この宣言は，変数の宣言に **[定数式]** を追加したものである．定数式というのは，演算結果が定数になる式である．具体的には，正の整数を指定すればよい．0や負の整数を指定することはできない．浮動小数点数もだめである．

次に例を示す．

```
int arr[10];
```

arrという名前のint型配列を宣言
arrは10個のデータを記憶できる

```
float price[100];
```

priceという名前のfloat型配列を宣言
priceは100個のデータを記憶できる

[ ] の中に書かれた定数式が，その配列で記憶できるデータの数を示す．この値をその**配列のサイズ**と呼ぶことにする．

たとえばint arr[10];のように宣言された配列arrは10個のint型データを記憶でき

るので，配列のサイズは10である．また，float price[100];のように宣言された配列priceは100個のfloat型データを記憶できるので，配列のサイズは100である．

2.2◆要素とインデックス

配列に記憶できる個々のデータを**要素**という．それぞれの要素は，あたかも1つの変数のように振る舞う．値を代入したり，参照したりすることができる．

個々の要素は，**インデックス**（*index*）と呼ばれる整数で識別される．インデックスは，配列名の後ろに括弧 [] を付け，その中に指定する．

```
配列名[インデックス]
```

インデックスは0から順に振られる．たとえばint arr[10];のように宣言された配列arrは，次の10個の要素を変数のように使うことができる．

```
arr[0] ———— 1番目の要素
arr[1] ———— 2番目の要素
……
arr[9] ———— 10番目の要素
```

また，float price[100];のように宣言された配列priceは，次の100個の要素を変数のように使うことができる．

```
price[0] ———— 1番目の要素
price[1] ———— 2番目の要素
……
price[99] ———— 100番目の要素
```

いちばん最後の要素のインデックスは，宣言した数より1つ小さくなることに注意．インデックスは0から数えるからである．

2.3◆配列の使い方

次のサンプルプログラムは，配列を宣言し，各要素に値を代入し，それらの値を参照するまでの流れを示すものである．最初にリストを示し，その後でポイントを説明する．

【サンプルプログラム】

これは，配列の基本的な使い方を示すものである．3つの要素を持つ配列を宣言し，各要素に値を代入した後，その値を表示する．

```
// 配列の宣言と参照

#include <iostream.h>

int main()
{
    int price[3];

    price[0] = 19800;
    price[1] = 12800;
    price[2] = 9800;

    cout << "商品1の価格 = " << price[0] << endl;
    cout << "商品2の価格 = " << price[1] << endl;
    cout << "商品3の価格 = " << price[2] << endl;

    return 0;
}
```

まず，次の文で配列priceを宣言している．

```
int price[3];
```

配列priceの要素数は3である．priceの各要素はint型のデータを記憶できる．

次の3つの文で，配列の各要素に初期値を代入している．

```
price[0] = 19800;
price[1] = 12800;
price[2] = 9800;
```

最後に，次の3つの文で配列の各要素の値を表示している．

```
cout << "商品1の価格= " << price[0] << endl;
cout << "商品2の価格= " << price[1] << endl;
cout << "商品3の価格= " << price[2] << endl;
```

このサンプルプログラムの実行例を次に示す．

【実行例】

```
% a.out
商品1の価格 = 19800
商品2の価格 = 12800
商品3の価格 = 9800
% _
```

3 定石

配列の基本的な使い方は決まっている．この節では，配列を使う際のいろいろな定石を紹介する．

3.1◆初期値を与える

第2節のサンプルプログラムは，商品の価格を配列に保存した．このように配列の初期値は，あらかじめ決まっていることが多い．そのようなとき，配列を宣言すると同時に初期値を与えることができる．その方法を次に示す．

```
int price[] = { 19800, 12800, 9800 };
```

これは，int型の配列priceを宣言している．通常の宣言と異なるのは，[] の中に配列のサイズを書いていないことである．代わりに = 以下の部分が続いている．

{} の中には，各要素に与える初期値を順にカンマで区切って並べている．この結果，priceの各要素は次のように初期化される．

```
price[0] = 19800;
price[1] = 12800;
price[2] = 9800;
```

この宣言では，配列のサイズを省略しているが，{} の中に並べた初期値の数が配列のサイズになる．つまり，この宣言は次と同等である．

```
int price[] = { 19800, 12800, 9800 };
```

↓ 次と同等

```
int price[3];           // 配列を宣言し

price[0] = 19800;       // 初期値を与える
price[1] = 12800;
price[2] = 9800;
```

C++は自由書式の言語なので，単語以外のところなら自由に改行することができる．本書では，この宣言を次のように書くことにする．

```
int price[] = {
    19800,
    12800,
    9800
};  ―――――― セミコロンを忘れない
```

このように書くことにより，配列の初期値が見やすくなる．このように書くとき，最後の } の後のセミコロンを忘れやすいので注意！

この宣言のしかたで第2節のサンプルプログラムを書き換えたものを次に示す．

【サンプルプログラム】

```
//配列の宣言と参照

#include <iostream.h>

int main()
{
    int price[] = {
        19800,
        12800,
        9800
    };  ―――――――――― ここにはセミコロンが必要

    cout << "商品1の価格 = " << price[0] << endl;
    cout << "商品2の価格 = " << price[1] << endl;
    cout << "商品3の価格 = " << price[2] << endl;

    return 0;
}
```

【実行例】

```
% a.out
商品1の価格 = 19800
商品2の価格 = 12800
商品3の価格 = 9800
% _
```

3.2◆配列のサイズ

配列のサイズ――すなわち配列の要素数が必要になることがある．第7章「**標準入力**」で説明した**sizeof**演算子を使うと，配列のサイズを求めることでができる．ただし，少し演算を行う必要がある．

まず，sizeof演算子を次のように使って，配列全体のサイズを求める．

sizeof 配列名 ―――― (1)

これは，配列全体のサイズであって，配列のサイズ（配列の要素数）ではないことに注意．配列全体のサイズというのは，その配列の全要素がメモリ上に占める大きさ（バイト数）のことである．

次に，配列の要素1つ分のサイズ（メモリ上のサイズ）を求める．

sizeof 配列名[0] ―――― (2)

0は，配列の要素を指定するインデックスである．0以外のインデックスを指定しても同じであるが，インデックス0なら，どの配列にも存在する．

最後に（1）÷（2）を計算する．これが求める配列のサイズである．

sizeof 配列名 / sizeof 配列名[0]

【サンプルプログラム】

このサンプルプログラムは，配列全体のサイズ，1要素分のサイズ，要素数を求めて表示するものである．

```
#include <iostream.h>

int main()
{
    int arr[100];
```

```
    cout << "全配列のサイズ = " << sizeof arr << endl;
    cout << "要素のサイズ = " << sizeof arr[0] << endl;
    cout << "要素の数 = " << sizeof arr / sizeof arr[0] << endl;

    return 0;
}
```

【実行例】

```
% a.out
全配列のサイズ = 400
要素のサイズ = 4
要素の数 = 100
% _
```

3.3◆全要素の出力

配列の全要素を出力するには，for文を使う．そのとき，前の節で説明した方法で配列のサイズを求め，それを繰り返しの数に指定する．

```
for (int i = 0; i < sizeof arr / sizeof arr[0]; i++)
    cout << arr[i] << endl;
```

【サンプルプログラム】

このサンプルプログラムは，for文を使って配列の全要素を表示する．

```
#include <iostream.h>

int main()
{
    int price[] = {
        19800,
        12800,
        9800
    };  ―――― ここにはセミコロンが必要

    for (int i = 0; i < sizeof price / sizeof price[0]; i++)
        cout << "商品" << i << "の価格 = "
             << price[i] << endl;

    return 0;
}
```

【実行例】

```
% a.out
商品0の価格 = 19800
商品1の価格 = 12800
商品2の価格 = 9800
% _
```

3.4◆全要素に同じ値を設定する

配列の初期値として，すべての要素に同じ値を設定したいことがある．そのときは，前の節でも使ったfor文で配列の初期化を行う．

【サンプルプログラム】

次のプログラムは，配列の全要素を0に初期化し，次にその値を表示している．

```
#include <iostream.h>

int main()
{
    int arr[7];

    // 全要素に0を代入する
    for (int i = 0; i < sizeof arr / sizeof arr[0]; i++)
        arr[i] = 0;

    // 全要素を表示する
    for (int i = 0; i < sizeof arr / sizeof arr[0]; i++)
        cout << "arr[" << i << "] = "
             << arr[i] << endl;

    return 0;
}
```

【実行例】

```
% a.out
arr[0] = 0
arr[1] = 0
arr[2] = 0
arr[3] = 0
arr[4] = 0
arr[5] = 0
arr[6] = 0
% _
```

4 応用

最後に配列の典型的な応用例を示す．このプログラムは，また配列が持つ限界を示すものでもある．

4.1◆入力データの保存と加算

プログラムは，複数データの合計を求めるという簡単なものである．データを合計するだけなら，普通の電卓でもできる．このプログラムは，電卓のようにただ加算するだけでなく，入力したデータの記録を残す．このため，データが正しく入力されたかをチェックすることができる．

このプログラムは，ループ変数を利用しているので，わかりにくい面もある．最初に，リストとその実行例を示す．その後でプログラムの詳しい説明を行う．

【サンプルプログラム】

これは，複数データの合計を求めるプログラムである．

最初にデータを入力するように要求される．合計を求めたいデータを入力する．負数を入力すると入力を終了する．最後に，入力したデータと，データの数，および合計が表示される．

```
// 複数データの合計を求める
// 負の数を入力すると入力の最後と見なす
// 処理可能なデータ数は100

#include <iostream.h>

int main()
{
    const int MAX = 100;
    int i, d[MAX], ndata, sum = 0;

    cout << "複数データの合計を求めます" << endl
         << "入力の最後は負数を入力してください" << endl;

    for (i = 0; i < MAX; i++) {
        cout << "データ" << i + 1 << " ? ";
        cin >> d[i];
        if (d[i] < 0)
```

```
            break;
    }
    ndata = i;

    cout << endl;
    for (i = 0; i < ndata; i++) {
        cout << "データ" << i + 1 << " = " << d[i] << endl;
        sum += d[i];
    }
    cout << endl
         << "データ数= " << ndata << endl
         << "合計= " << sum << endl;

    return 0;
}
```

【実行例】

```
% a.out
複数データの合計を求めます
入力の最後は負数を入力してください
データ1 ? 32571 ―――――――――― データを入力していく
データ2 ? 264
データ3 ? 0
データ4 ? 8507
データ5 ? 493
データ6 ? -1 ―――――――――― 入力の終了は負数

データ1 = 32571 ―――――――――― 入力したデータを確認できる
データ2 = 264
データ3 = 0
データ4 = 8507
データ5 = 493

データ数= 5 ―――――――――― 結果の表示
合計= 41835
% _
```

次に，このプログラムのポイントを説明していく．

4.2◆データの最大数

このプログラムは，入力可能なデータ数に制限がある．

```
const int MAX = 100;
```

MAXが入力可能なデータの最大数を表している．この値は，プログラムの最後まで変わらないので，定数として宣言している．MAXは，配列の宣言にも使われている．

```
int d[MAX];
```

定数MAXの値を変えれば，入力可能なデータ数を変更することができる．

4.3◆変数

このプログラムで使用している各変数の意味を次に示す．

変数	用途
d	入力データを記憶する 複数のデータを記憶するので配列にしてある
ndata	実際に入力されたデータ数
sum	データの合計を計算する

そのほか，for文のループ変数として"i"を使用している．この変数は，最初のループを抜けたとき，実際に入力されたデータ数を表している．それをループの外でndataに代入している．

```
for (i = 0; i < MAX; i++) {
    …………
}
ndata = i;  ← ここ
```

このため，変数の宣言をforの中ではなく，プログラムの先頭のところで行っている．

4.4◆データの入力

このプログラムは，複数のデータを保存する必要があるので，そのための変数として配列を使っている．複数のデータを入力するのでfor文を使う．配列に対する典型的なやり方である．1回繰り返すごとにデータを1つ入力する．

```
for (i = 0; i < MAX; i++)
    cin >> d[i]; ←――――ここでデータを入力する
```

入力されたデータは，配列の各要素d[0], d[1],…, d[MAX - 1]に記憶される．

データを入力するとき，現在何件目の入力中であるかを表示するとわかりやすい．そこで，次の文を追加する．

```
for (i = 0; i < MAX; i++) {
    cout << "データ" << i + 1 << " ? "; ←――――この文を追加
    cin >> d[i];
}
```

4.5◆ループの脱出 —— break

入力は，forループの最後まで到達すると自動的に終了する．ただし，負数が入力された場合にも終了しなければならない．そのためには，for文の途中から脱出する方法を覚えなければならない．

この目的のためには，switch文のところでも使用したbreak文が使われる．break文をfor文の中に使うと，そこでfor文を脱出する．たとえば上のようなfor文があった場合，**文1**→ **文2**を実行したところでfor文を抜ける．**文3**は実行されない．

```
for (…) {
    文1
    文2
    break;
    文3
}
```

これでは，1回しかループの本体を実行しないので，通常はif文とbreak文を組み合わせて，ある条件を満たした場合だけループを抜けるようにする．たとえば，

次の場合は，変数xの値が正になったらループを抜ける．

```
for (…) {
      文1
      文2
      if (x > 0)
     break;
}
```

このサンプルプログラムの場合は，負数が入力されたらループを抜ければよいので，次のようになる．

```
for (i = 0; i < MAX; i++) {
      cout << "データ" << i + 1 << " ? ";
      cin >> d[i];
      if (d[i] < 0) ―――――――――― 負数が入力されたら
     break; ――――――――――――――― ループを抜ける
}
```

4.6◆データの件数

このプログラムでは，実際に入力されたデータ件数を保存しておく必要がある．後で，その件数分のデータを表示しなければならないからである．

最初のforループを抜けた時点で，ループ変数iの値が入力されたデータの件数を示している．なぜか？

```
for (i = 0; i < MAX; i++) {
      // ここでデータを入力する
      if (負数が入力された)
           // ループを抜ける
}
```

まず，負数が入力されずforループが最後まで実行されたとする．その場合，変数iの値はMAXになっている．これは，入力されたデータ件数そのものにほかならない．

次に負数が入力され，途中でループを抜けた場合を考える．この場合，d[i]には負の数が入っている．ということは，有効なデータはd[i - 1]までである．この場合

も，iは入力されたデータの件数を示している．なぜなら，データはd[0]から入力されているからである．d[0],…, d[i - 1]のデータ件数はiになる．

以上のように最初のforループを抜けた時点でループ変数iの値が入力されたデータの件数を示している．それを別の変数ndataに保存しておく．なぜなら，変数iは次のfor文でも使われるので，保存しておかないと入力されたデータ件数の値を失ってしまうからである．

```
for (i = 0; i < MAX; i++) {  ――――― 最初のfor文でデータ入力
      ............
}
ndata = i;  ――――― データ件数を保存

for (i = 0; i < ndata; i++) {  ――――― またiが使われる
      ............
}
cout << endl
      << "データ数= " << ndata << endl ――― データ件数を出力
      << "合計= " << sum << endl;
```

4.7◆合計の計算

変数sumに合計を作っている．計算に入る前に変数sumを0にクリアしておく．これは，変数宣言と同時に初期化するとよい．

```
int sum = 0;
```

合計を計算する部分も典型的な配列の応用になっている．合計すべきデータが配列の要素に入っているので，for文で反復しながら合計を取っていく．

このプログラムでは，2つのfor文を使っている．そのどちらで合計を計算してもよいが，ここでは2つ目のfor文でデータを表示しながら計算している．

```
for (i = 0; i < ndata; i++) {
      cout << "データ" << i + 1 << " = " << d[i] << endl;
      sum += d[i];  ◀――――― ここ
}
```

for文を抜けたところでsumに合計が出来上がっている．

4.8◆配列の限界

複数のデータを保存できるという点で配列は強力である．しかし，配列にも限界はある．それは，プログラム時に配列のサイズを決めておかなければならないということである．

たとえば，この節で取り上げたサンプルプログラムは，処理できるデータ数が100に制限されていた．それは，配列のサイズが100だからである．

```
int d[MAX];
```

MAXは100

大きな配列を確保しておけば，配列が不足することはない．しかし，大きな配列を使うと，それだけメモリを使用する．したがって，適当なサイズの配列を確保するのが普通である．たいていは，それでうまくいく．

しかし，時には実行中に配列の数が足りなくなることもある．そのときは，手遅れである．コンパイルし直さなければならない．

このように便利な配列ではあるが，そのサイズをソースプログラムを書くときに決めなければならないというのは痛い．

> **注意**　配列を使う代わりに「動的メモリ」を使う方法がある．少しプログラムが面倒になるが，実行時にデータ領域を増減することができる．

第16章 ポインタ

この章から**ポインタ**についての話が始まる．ポインタはC++の難関の1つである．ポインタがわからないと，C++の最大の特色であるクラスを理解することはできない．

ポインタは，変数の一種である．それも特殊な変数である．通常の変数は，100とか3.14とかといった具体的なデータを記憶する．それに対して**ポインタ**は，**アドレス**を記憶する．では，アドレスとは何だろう？

ポインタを理解するには，アドレスを知る必要がある．まず，そこらあたりの話から始める．

1 アドレス

アドレスというのは，メモリ（記憶装置）の位置を示すハードウェア上の数値である．

メモリは，個々のデータを1バイト単位で記憶する．**バイト**というのはデータの大きさを示す単位で，2進数8個分の大きさである．つまり，1つのメモリは2進数を8個記憶することができる．

その1バイト単位のメモリを識別するための数値が**アドレス**である．

CPUがメモリを読み書きするときは，アドレスを指定して行う．CPUは，次のような機械語の命令を実行しながらメモリを読み書きしている．

図16-1▶

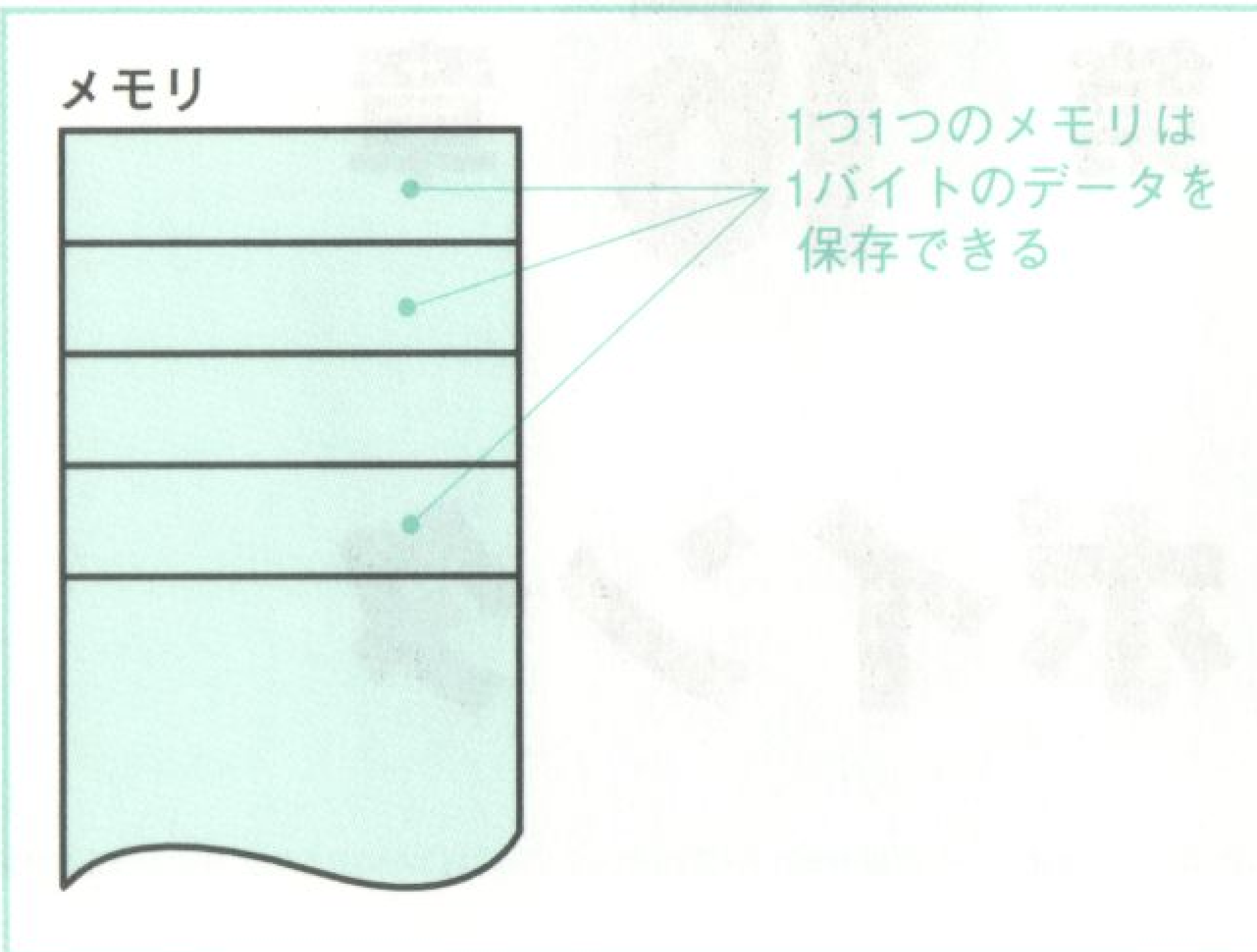
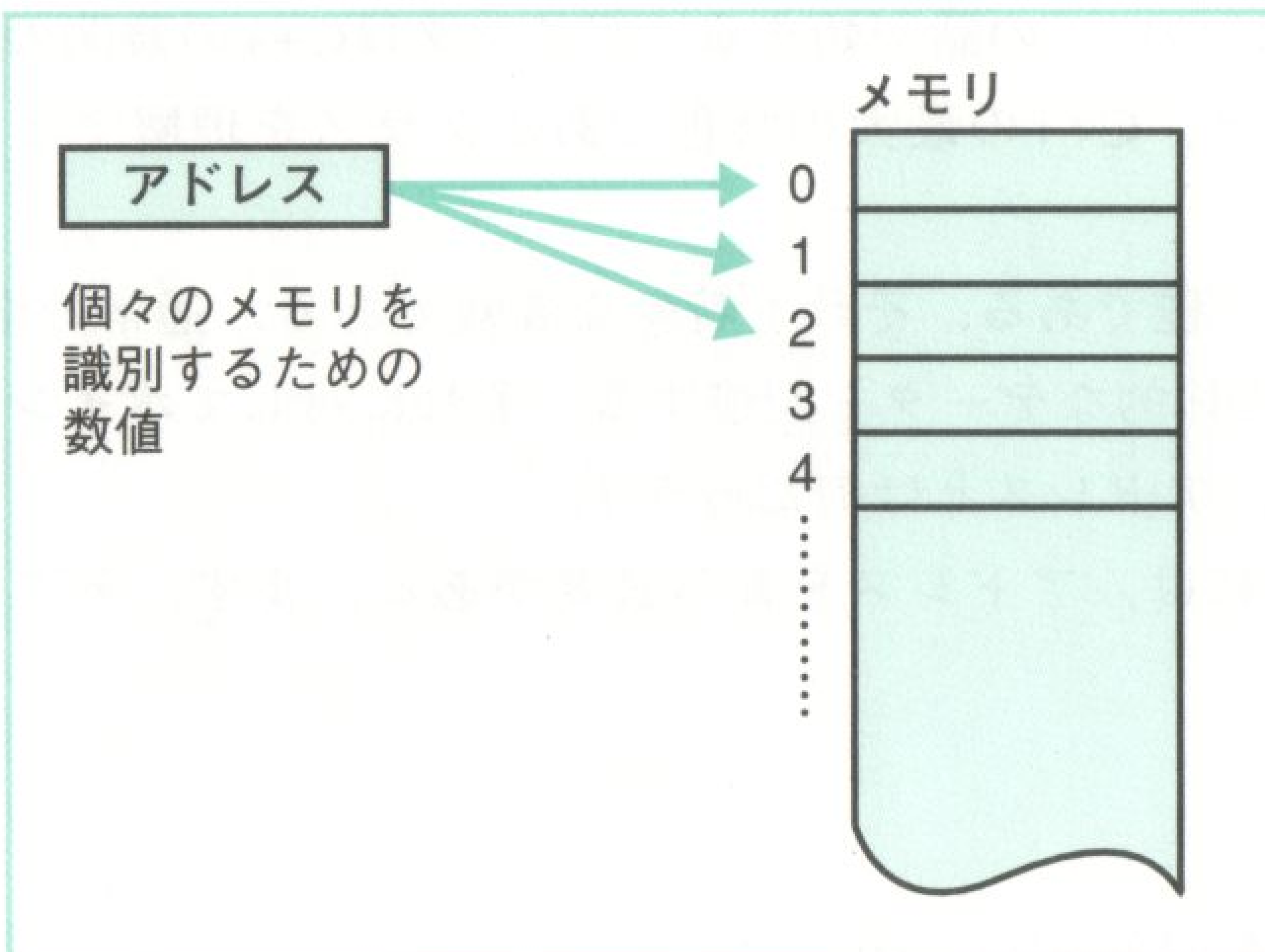


図16-2▶



- アドレス10に0xffを書き込め
- アドレス11のデータを読み出し，それをレジスタ（CPU内の記憶装置）に格納せよ

ポインタが記憶するのは，このメモリ上の位置を示すアドレスである．

はじめてポインタを学ぶと，なぜそのようなものが必要なのか理解しにくい．メモリのアドレスを記憶するのに何の価値があるのかわからない．しかし，高度なプログラミングをするには，ポインタが必要である．

次の節では「なぜポインタが必要か？」については，ひとまず目をつぶり，ポインタの使い方の基本を説明する．その後で，なぜポインタが必要なのかを説明する．

2 ポインタの基本

この節では，ポインタの使い方の基本を説明する．

2.1◆宣言

ポインタは変数の一種である．したがって，ポインタを使うには，他の変数同様に宣言が必要である．宣言のしかたは，ほとんど普通の変数と同じであるが，変数名（ポインタ名）の前にアスタリスク（*）を付ける．この“*”は，乗算を意味する演算子ではないので注意．

```
int n;  ―――― nは普通の変数
int *n; ―――― *が付いているのでnはポインタ
```

まとめると，ポインタの宣言は次のようになる．

```
  ┌―――― そのポインタが記憶する変数の型を指定する
  ↓
型 *ポインタ名;
```

ここでポインタには，intとかfloatといった型があることに注意．ポインタは，アドレスを記憶する．アドレスに型があるというのは妙である．ポインタの型とは，何を意味するものだろう？

> **注意**　ポインタの名前はpvalとかvalpのように，先頭または末尾にpを付けることが多い．こうすることにより，名前を見ただけでポインタであることがわかる．
>
> しかし，これは絶対的な規則というわけではないので，自由な名前を付けてかまわない．

2.2◆変数のサイズ

ポインタの型を理解するため，ここで，いろいろな型の変数がメモリ上でどのくらいのサイズを使うかを調べる．

「アドレス」のところで説明したように，メモリの単位は1バイトである．1バ

イトは，2進数が8個分の大きさである．

$$2^8 = 2\times2\times2\times2\times2\times2\times2\times2$$
$$= 256$$

この計算により，1つのメモリでは256種類のデータしか区別できないことがわかる．たとえば正の整数を1バイトに記憶したとすると，0～255まで256種類しか扱えない．そこで，たいていのデータ型は，複数のメモリを使って表現する．

あるデータ型が何バイトのメモリを使うかは，これまで何回か登場したsizeof演算子を使って調べることができる．

sizeof(型名)

たとえばint型のデータを調べるなら，次のようにする．

sizeof(int)

次のプログラムで，各データ型が何バイトを使うか調べてみる．

【サンプルプログラム】

このプログラムは，代表的なデータ型が何バイトのメモリを使うかを調べるものである．標準入力の章（第7章）で説明したものと同じである．

```
// sizeof.C -- 各データ型のサイズを求める

#include <iostream.h>

int main()
{
    cout << "char        : " << sizeof(char) << endl
         <<endl;

    cout << "short       : " << sizeof(short) << endl;
    cout << "int         : " << sizeof(int) << endl;
    cout << "long        : " << sizeof(long) << endl
         << endl;

    cout << "float       : " << sizeof(float) << endl;
    cout << "double      : " <<sizeof(double) << endl;
    cout << "long double : " <<sizeof(long double) << endl;

    return 0;
}
```

【実行例】

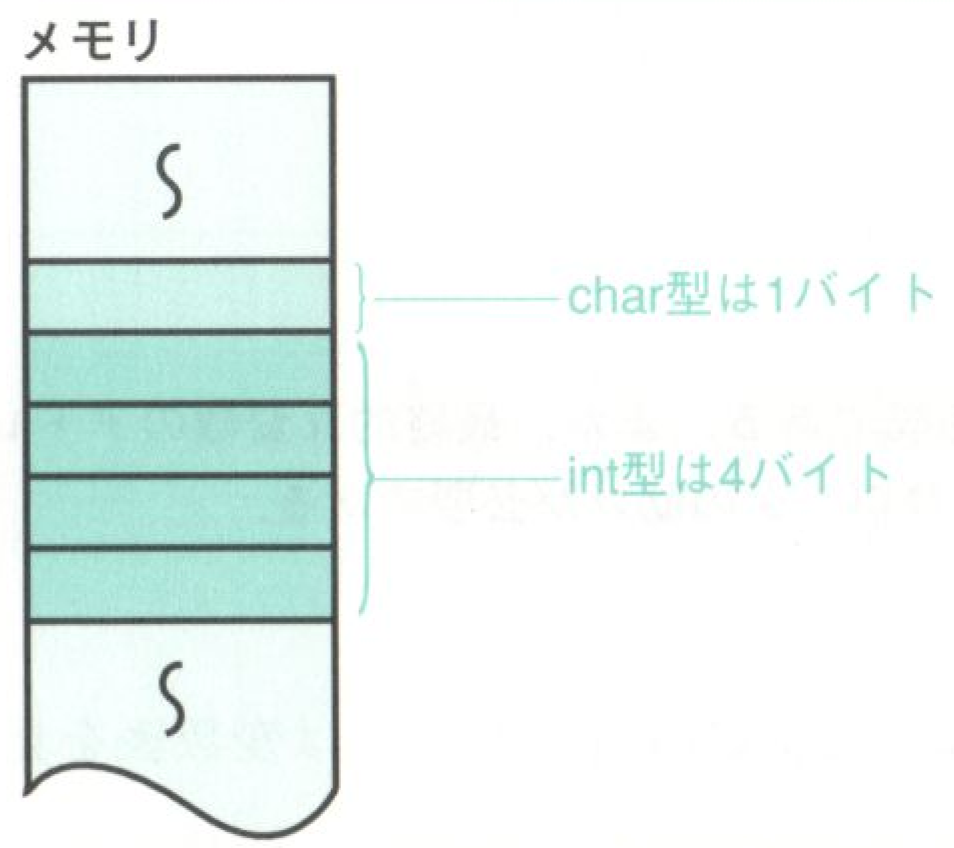
```
% a.out
char        : 1 ──────────── 文字型のサイズ

short       : 2 ──────────── 整数型のサイズ
int         : 4
long        : 4

float       : 4 ──────────── 浮動小数点型のサイズ
double      : 8
long double : 12
% _
```

この結果を見ると，たとえばint型なら4バイト，double型なら8バイトのメモリを使うことがわかる．このように，データ型によって，使用するメモリの大きさは異なる．

図16-3▶



※データ型によって使用するメモリの量は異なる

注意　各データ型が使用するメモリのサイズは，処理系（コンパイラおよびその使用環境）によって異なる．

2.3◆変数表

ポインタの型を理解するためは，もうひと踏んばり必要である．次の文をC++がどのように解釈するかを見てみよう．

```
cout << x << endl;
```

この文をコンパイルするためには，C++本来の機能にプラスすることのiostreamパッケージが必要である．そのことは横に置いておいて，C++はこの文を構文解釈することにより，変数xの値をcoutに渡すことを理解する．

そのためには，変数xの値を調べる必要がある．変数xの値は，メモリの上のどこかにある．そのどこかがわからなければ，xの値もわからない．

変数がどのアドレスに記憶されるかは，変数の宣言のとき決められる．C++コンパイラは，変数宣言を見つけると，その変数のためのアドレスを決定し，それを**変数表**に記録する．たとえば，次のような変数宣言があったとする．

```
int x;
float y;
```

C++コンパイラは，この宣言を見て次のような変数表を作成する．

変数名	アドレス
x	1000番地
y	1004番地

注意　この変数表はまだ未完成である．また，最終的な変数のアドレスを決定するためには，リンカおよびローダの助けが必要である．

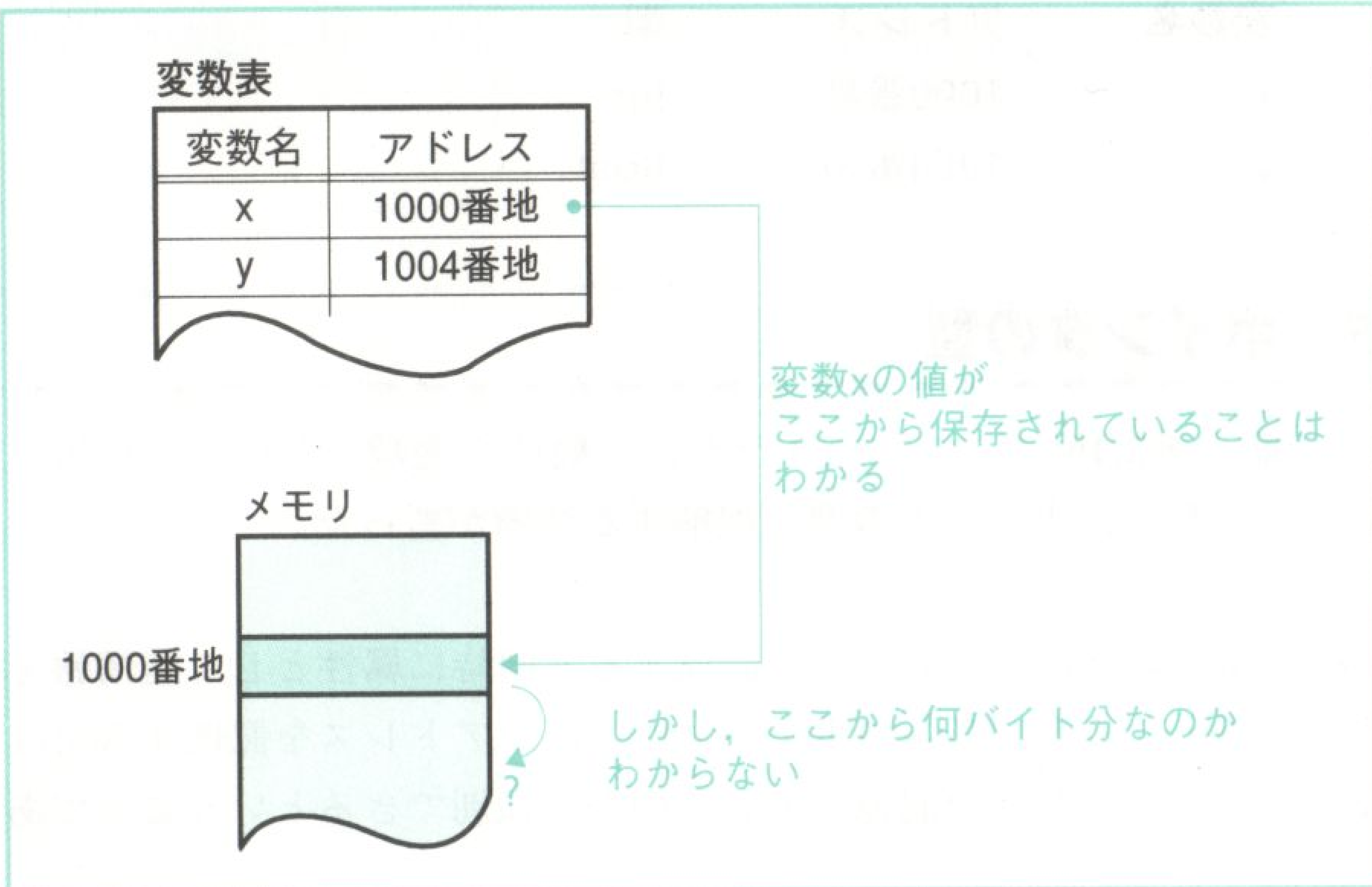
逆に変数のアドレスが必要になると，C++コンパイラは変数表を見て，そのアドレスを知る．

2.4◆変数の型

上の注意でも指摘したように，いま示した変数表は未完成である．なぜなら変数のサイズが記録されていないからである．「**変数のサイズ**」のところでも指摘したように，変数の型によって，使用するメモリの量が異なる．

いまの変数表では，変数の先頭アドレスはわかるが，そこから何バイトを占めているのかがわからない．

図16-4▶



それならば，変数表に変数のサイズも加えればよいように思われる．

変数名	アドレス	サイズ
x	1000番地	4
y	1004番地	4

しかし，これでもまだ不足である．なぜなら，型が違っても使用するサイズが同じ場合もあるからである．

「**変数のサイズ**」のところで示したサンプルプログラムの実行例を見るとわかるが，たとえばintもlongもfloatも同じ4バイトである．intもfloatも同じ4バイトを使用するが，その使い方が異なる（整数と実数でメモリの使い方が同じということはない）．したがって，使用するバイト数がわかっても，変数の値を取り出すことができないのである．

メモリから変数の値を取り出すには，次の2つの情報が必要である．

- **その変数が格納されている先頭アドレス**
- **その変数の型（メモリの使い方）**

変数の型がわかれば，その変数が使用するメモリのサイズも使い方もわかる．したがって，変数表には，次の項目を記録する必要がある．

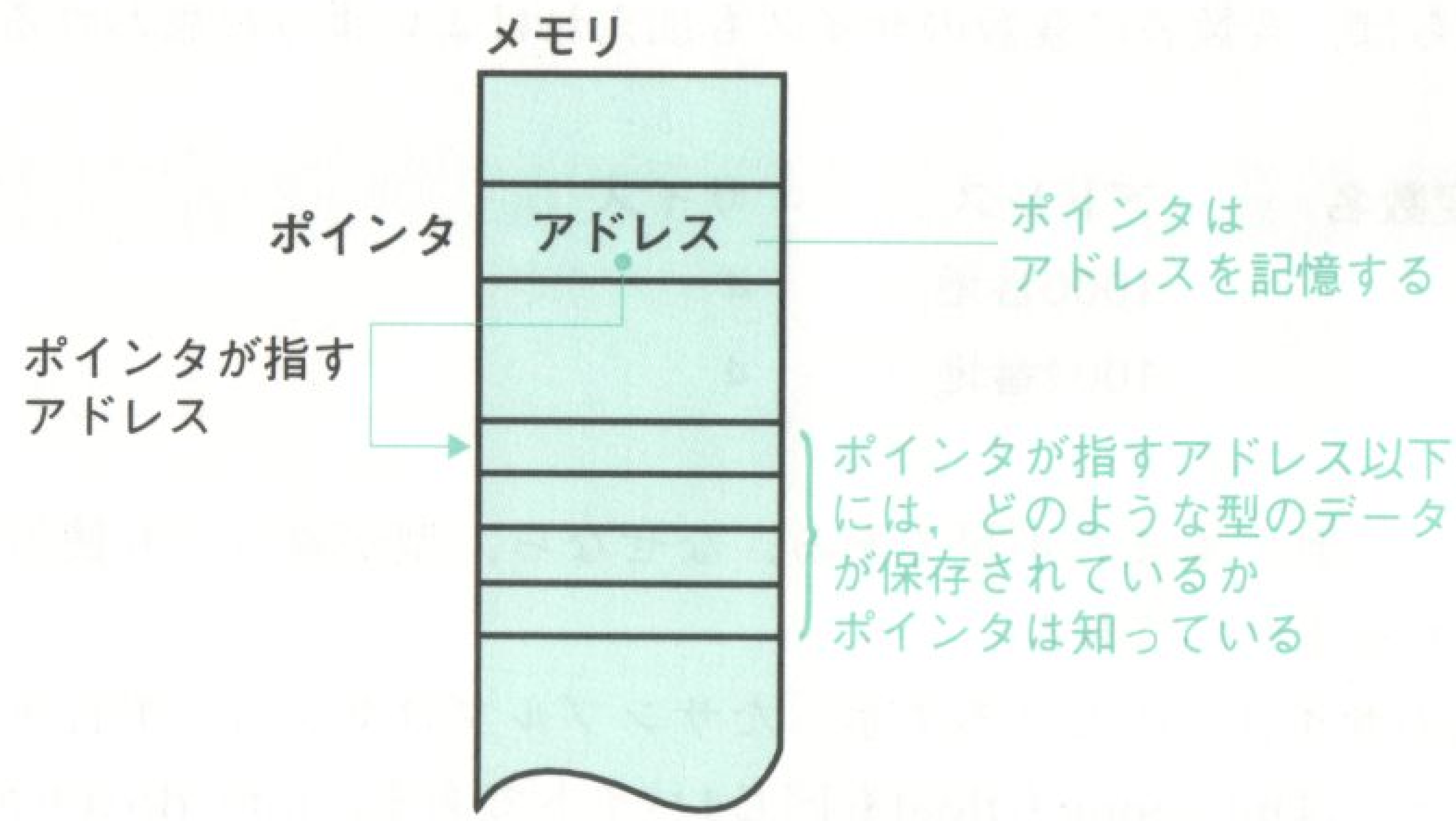
変数名	アドレス	型
x	1000番地	int
y	1004番地	float

2.5◆ポインタの型

第2.2節～第2.4節で，コンパイラが，一般的な変数をどのように扱うかを説明してきた．これで，ポインタの型を理解する準備が整った．

C++のポインタは，アドレスを記憶すると同時に属性として**型情報**を持つ．どういうことかというと，C++のポインタは，ただアドレスを記憶するだけでなく，どんなデータのアドレスを記憶しているのかを識別できるということである．

図16-5▶



なお，ポインタが記憶するアドレスと，その型情報が記録されている場所は異なる．アドレスの方は，そのままポインタが記憶する．しかし，型情報の方は変数表に記録される．たとえば，次のようなポインタの宣言があったとする．

```
int *n;
```

この場合，変数表には次のように記録される．

変数名	アドレス	型
n	1000番地	int *（int型のポインタ）

2.6◆代入

ポインタは変数の一種なので，ポインタにデータを記憶させるには = などの代入演算子を使用する．ただし，記憶させるデータはアドレスに限る．

次に例を示す．

```
int *np;
np = 0xbffffd50;
```

この場合，ポインタnpにbffffd50（このアドレスは16進数で示している）というアドレスを代入している．

2.7◆アドレス演算子

ポインタは，メモリのアドレスを代入することができる．したがって，必要ならポインタに他の変数のアドレスを記憶させることもできる．

変数の前に**アドレス演算子&**を付けると，その変数の値ではなく，変数の先頭アドレスを意味するようになる．

```
n  ―――――― これは変数nの値
&n ―――――― これは変数nの先頭アドレス
```

したがって，次のようにすれば，ポインタに他の変数のアドレスを記憶させることができる．

```
int val;   ―――――― 整数型の変数valを宣言
int *pval; ―――――― 整数型のポインタpvalを宣言

pval = &val; ―――― ポインタpvalに変数valのアドレスを代入
```

2.8◆逆参照

ポインタに他の変数のアドレスが記憶されているとき，その変数の値を取り出すことができる．これは，**逆参照演算子**と呼ばれる単項演算子*を使って行う．

注意　アスタリスクは，乗算，ポインタの宣言，逆参照演算子のようにいろいろな場面で使われるので，混乱しないこと．

次に例を示す．

```
int val = 100; ―――――― 整数型の変数valを宣言
                        同時に初期値100を代入する
int *pval; ―――――――― 整数型のポインタpvalを宣言

pval = &val; ――――――― ポインタpvalに変数valのアドレスを代入
cout << *pval << endl;
```

最後の文で*pvalを出力している．pvalは，ポインタである．ポインタの前に * が付いているので * は逆参照演算子である．この場合，pvalは変数valのアドレスを記憶しているので，valの値100が出力される．

以上を，次のコンパイル可能なプログラムでテストする．

【サンプルプログラム】

```
#include <iostream.h>

int main()
{
    int val = 100;
    int *pval;

    pval = &val;
    cout << *pval << endl;

    return 0;
}
```

【実行例】

```
% a.out
100 ―――――――――――――――――― これは変数valの値
% _
```

逆参照演算子を使って他の変数の値を取得するとき，ポインタの型情報が使われる．C++コンパイラが*pvalを解釈する手順を次に示す．

① ポインタpvalが記憶しているアドレスを取り出す
② 変数表からポインタpvalの型を調べる
　仮にそれがintであるとする
③ ①で取得したアドレス以下に②で取得したintのデータを取り出す

3 参照引数

第2節でポインタの基本を説明した．この節からは，実際にポインタが使われる場面を見ながら，なぜポインタが必要なのかを説明していく．

この節は，ポインタの最初の応用例として参照引数を説明する．

3.1◆swap関数

いま，2つのint型変数の値を交換するswapという関数を作りたいとする．swapは次のように使用する．

```
n1 = 100;
n2 = 200;

swap(n1, n2);
```

——— n1とn2の値が交換される
その結果，n1 = 200, n2 = 100になる

ここまでの知識で関数swapを作成すると次のようになる（実はこのコードにはバグがある）．

```
void swap(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

いきなりx = yを実行するとxの値が消えてしまう．そこで，一時的な変数tempを用意する．xの値をtempに保存し，yの値を代入する．その後，tempの値をyに代入すれば，xとyの値を交換することができる．

注意　一時変数を使わない方法を第3.6節で説明する．

この関数をテストするため，次のサンプルを実行する．

【サンプルプログラム】

```
#include <iostream.h>

// xとyの値を交換する(第1版)
void swap(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int n1 = 100, n2 = 200;

    cout << "n1, n2の最初の値 : " << n1 << ", " << n2 << endl;
    swap(n1, n2);
    cout << "swap実行後の値    : " << n1 << ", " << n2 << endl;

    return 0;
}
```

【実行例】

```
% a.out
n1, n2の最初の値 : 100, 200
swap実行後の値    : 100, 200 ―――――― 値が交換されていない!!
% _
```

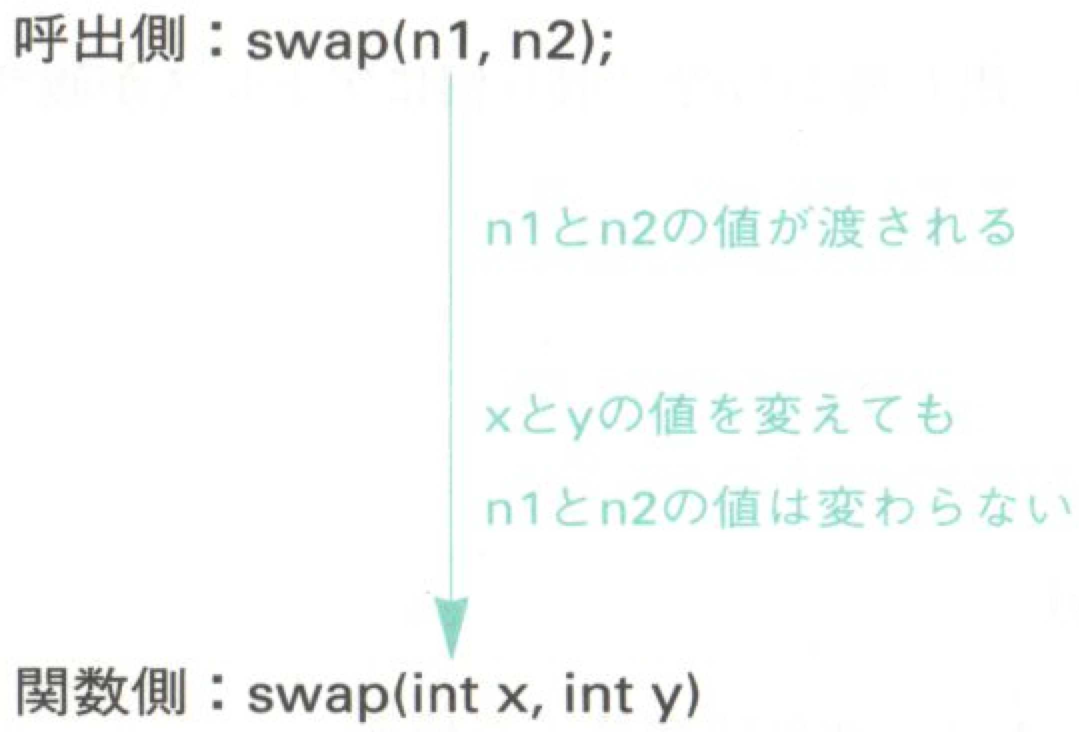
この実行結果を見てみると，n1とn2の値は交換されていないことがわかる．残念ながら，ここで書いたswapにはバグがある．

3.2◆値引数

第3.1節で書いたswapがうまく動作しなかったのは，C++における「引数の仕組み」に原因がある．

C++は，関数の内部を外部からできるだけ隔離しようとする．関数内部で行った処理を，関数の外部にできるだけ影響を与えないようにしようとする．プログラマーが関数内部のコードに集中できるようにするためである．

そのため，C++の引数は**値引数**といって，実引数の値が仮引数にコピーされる仕掛けになっている．この方式では，たとえ関数の内部で仮引数の値を変更しても，実引数には影響を与えない．



C++では，実引数（関数を呼び出すときに指定する引数）と仮引数（関数側で定義した引数）とは何の関係もない．したがって，いくら関数の中でxとyの値を交換しても，呼び出し側のn1，n2には何の影響も及ぼさない．

第3.1節のswapがうまく機能しなかったのは，このためである．swap内部で値を交換しても，呼び出し側の変数の値を交換することはできない．

関数の中で実引数の値を変更したければ，**ポインタ引数**を使う必要がある．

3.3◆ポインタ引数

ポインタ引数というのは，文字どおりポインタを引数にしたものである．

ポインタ

```
void swap(int *x, int *y)
{
        ............
}
```

仮引数xとyの前に * を付けてポインタにしている．このように定義された関数を呼び出すときは，引数にもポインタまたはアドレスを指定する必要がある．

```
swap(&n1, &n2);
```

n1とn2の前にアドレス演算子&を付けている
この結果，n1とn2のアドレスが渡される

ポインタ引数を使うと，関数の呼び出し側から受け取り側にアドレスが渡される．

呼出側：swap(&n1, &n2);

n1とn2のアドレスが渡される

関数側：swap(int *x, int *y)

関数側で実引数の値を参照するときは，逆参照演算子を使う．

x ―― これは実引数のアドレス
*x ―― これが実引数の値

このとき，関数の中で実引数の値を変更したらどうなるか？

```
*x = 100;
```

*xは実引数の値が実際に保存されているメモリを指しているので，実引数の値が変更される．

このように，ポインタ引数を使うと，逆参照演算子を使って実引数の値を直接参照できるようになる．この結果，関数内部で呼び出し側の引数の値を変更できるようになる．

3.4◆swap関数(第2版)

swapをポインタ引数を使うように変更して，正しく値が交換されるように修正する．新しいswapを次に示す．

```
void swap(int *x, int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```

【サンプルプログラム】

これは，swapの修正版と，その使い方を示すものである．

```
#include <iostream.h>

// xとyの値を交換する(第2版)
void swap(int *x, int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}

int main()
{
    int n1 = 100, n2 = 200;

    cout << "n1, n2の最初の値 : " << n1 << ", " << n2 << endl;
    swap(&n1, &n2);
    cout << "swap実行後の値   : " << n1 << ", " << n2 << endl;

    return 0;
}
```

【実行例】

```
% a.out
n1, n2の最初の値 : 100, 200
swap実行後の値   : 200, 100
% _
```

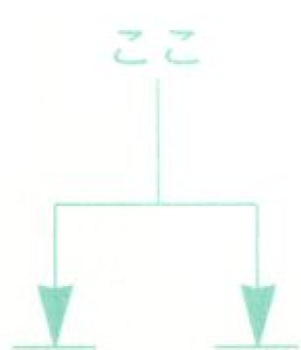
今度は，n1とn2の値が交換されている．

このように，ポインタ引数を使うと，関数内部で呼び出し側の引数の値を変更することができる．これがポインタを使う利点の1つである．

3.5◆参照引数

関数内部で実引数の値を変更したいときは，ポインタ引数を使う必要がある．しかし，ポインタ引数を使うと，必然的に逆参照演算子を使うことになり，関数内部が複雑化する傾向がある．また，その関数を使うとき，アドレスを渡さなければならないということを気に留めていなければならない．

そこで，ポインタ引数を簡略化して書く方法が用意されている．それが，**参照引数**である．参照引数を使うのは簡単である．ただ，引数の前に&を付けるだけである．




```
void swap(int &x, int &y)
```

これだけで，実引数の値を変更できるようになる．関数内部では，x，yを普通の変数のように使うだけである．逆参照演算子を使う必要はない．

```
void swap(int &x, int &y)
{
    int temp;

    temp = x;  ——— あたかも普通の変数のように使う
    x = y;
```

```
        y = temp;
    }
```

また，関数を呼び出すときもアドレスを渡す必要はない．普通に変数を指定すればよい．

```
    swap(n1, n2);
```

これでswapが正しく動く．

【サンプルプログラム】

これは，swapを参照引数を使って書き換えたものである．

```
#include <iostream.h>

// xとyの値を交換する(第3版)
void swap(int &x, int &y) ―――― 参照引数に変更した
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int n1 = 100, n2 = 200;

    cout << "n1, n2の最初の値 : " << n1 << ", " << n2 << endl;
    swap(n1, n2);
    cout << "swap実行後の値   : " << n1 << ", " << n2 << endl;

    return 0;
}
```

【実行例】

```
% a.out
n1, n2の最初の値 : 100, 200
swap実行後の値   : 200, 100
% _
```

3.6◆一時変数を使わない値の交換

この節は，脱線である．参照引数とは直接関係はない．もう1つのswap関数の作り方についてである．

2つの値を交換するswapでは，**一時変数**（臨時に使用する使い捨ての変数）tempを用意して，それを仲介に値を交換した．

```
void swap(int &x, int &y)
{
    int temp; ———— 一時変数

    temp = x;
    x = y;
    y = temp;
}
```

ここでのテーマは，一時変数を使わないでswapを実現する方法である．そのようなことが可能だろうか？　たとえば，一時変数を使わないでyの値をxに移すと，その瞬間，xのもとの値が消えてしまう．

```
x = y; ———— yの値をxに移すことはできるが
            xの値をyに移すことはできない
```

このため，どうしても一時変数を用意してxの値を保存する必要がありそうである．

一時変数を使わないでswapを実現する方法については，排他的論理和を使う方法が有名である．しかし，ここではもっと簡単な「引き算」を使う方法を説明する．

この方法のミソは，yの値をxに代入するのではなく，その差を代入することにある．

```
x = y - x;
```

この後は，新旧のx，yの値が入り交じってわかりづらくなるので，それぞれの旧の値をx0，y0とする．すると，現在x，yの値は次のようになっている．

```
x : y0 - x0
y : y0
```

この状態でyからxを引くと，次の計算により旧xの値が得られる．

```
y - x = y0 - (y0 - x0)
      = y0 - y0 + x0
      = x0
```

すなわち，次の式でyに旧xの値が移される．

```
y -= x;
```

現在，x，yの値は次のようになっている．

```
x : y0 - x0
y : x0
```

この状態でxにyを足すと，次の計算により旧yの値が得られる．

```
x + y = (y0 - x0) + x0
      = y0 - x0 + x0
      = x0
```

すなわち，次の式でxに旧yの値が移される．

```
x += y;
```

これで，一時変数を使わずにx，yの値を交換できることが証明された．新しいswapのリストを以下に示す．

```
void swap(int &x, int &y)
{
    x = y - x;
    y -= x;
    x += y;
}
```

【サンプルプログラム】

```
#include <iostream.h>

// xとyの値を交換する(第3版)
void swap(int &x, int &y)
{
    x = y - x;
    y -= x;
```

```
    x += y;
}

int main()
{
    int n1 = 100, n2 = 200;

    cout << "n1, n2の最初の値 : " << n1 << ", " << n2 << endl;
    swap(n1, n2);
    cout << "swap実行後の値   : " << n1 << ", " << n2 << endl;

    return 0;
}
```

【実行例】

```
% a.out
n1, n2の最初の値 : 100, 200
swap実行後の値   : 200, 100
% _
```

4 ポインタと配列

ポインタが使われるもう1つの場面としては，ユーザー定義型への利用がある．すなわち，クラスや配列のデータをポインタを通して操作する．この用途が最も重要である．

配列の各要素をポインタを通して操作する例を次の章で説明する．クラスについての応用は，第2部で説明する．

第17章 ポインタと配列

配列の名前は，配列要素が記憶されているメモリの先頭アドレスを表している．このため，配列名をあたかもポインタのように使うことができる．配列名をポインタに代入したり，ポインタを使って配列の要素を参照することが可能である．

この章では，配列とポインタの相互利用について説明する．また，配列とポインタの違いについて説明する．

1 配列名とアドレス

最初に配列の名前をポインタに代入し，そのポインタを逆参照するとどうなるかという例を示す．まず，次のように配列を用意する．

```
int d[] = {
    1111,
    2222,
    3333,
};
```

配列を宣言すると同時に初期値を与えている．この結果，この配列は次の3つの要素を持つことになる．

```
d[0] = 1111;
d[1] = 2222;
d[2] = 3333;
```

次にポインタを用意する．ポインタの型は，配列の型と同じにする．

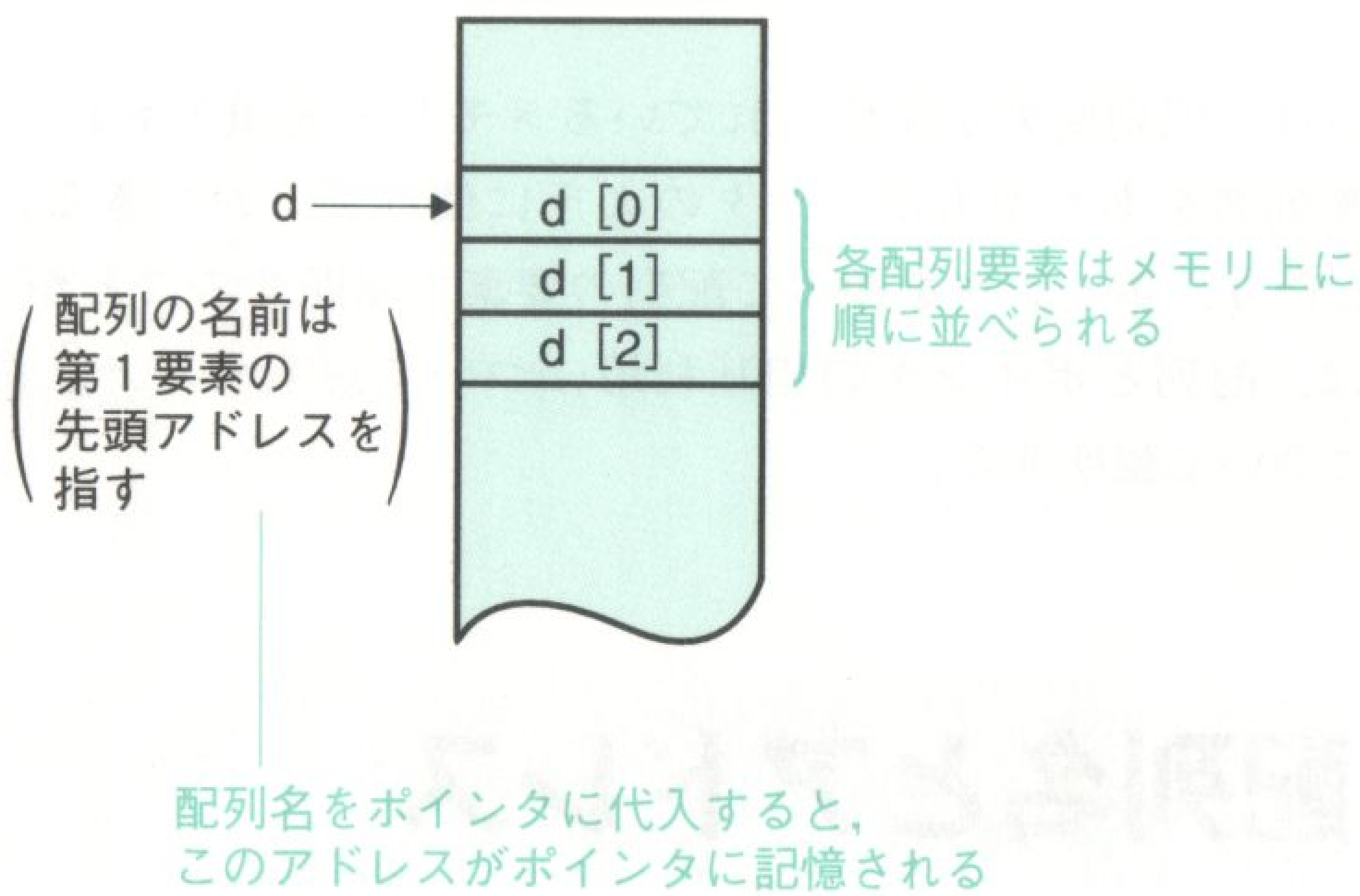
int *p;

そして，このポインタに配列の名前を代入する．

p = d;

配列の名前dは，配列の先頭アドレスを意味する．したがって，この代入により，ポインタpは，配列の最初の要素の先頭アドレスを記憶することになる．

図17-1▶



この状態でポインタpを逆参照すると，どうなるだろう？　すなわち*pは，どの値を返すだろう？　それを次のサンプルプログラムで確かめる．

【サンプルプログラム】

```
#include <iostream.h>

int main()
{
    int d[] = {
        1111,
        2222,
        3333,
    };
```

```
    int *p;

    p = d;                          配列名をポインタに代入
    cout << *p << endl;             ここでpを逆参照する

    return 0;
}
```

【実行例】

```
% a.out
1111                                これはd[0]の値
% _
```

この実行結果を見ると，*pは配列の最初の要素——すなわち，d[0]の値を返すことがわかる．

以上の結果をまとめると，次のようになる．

- **配列の名前dは，配列の先頭アドレスを意味する．**
- **このアドレスは配列の第1要素の先頭アドレスでもある．**
- **配列の名前をポインタに代入し，逆参照すると，第1要素の値を得ることができる．**

2 ポインタの演算

ポインタを使って配列の第1要素を参照できるようになった．第2要素以下を参照するには，ポインタについての演算を理解する必要がある．

ポインタは変数の一種ではあるが，アドレスを記憶するという特殊性があるため，可能な演算にも制限がある．また，その演算結果も（合理的ではあるが）予想とは反するものであったりする．

ポインタについては，次の演算が可能である．

- 整数値の加減
- アドレスとの比較
- 減算

いまpというint型のポインタがあって，その初期値（アドレス）が1000であるとする．

```
int p = 1000;
```

このポインタに1を加えるとどうなるだろう？

```
p++; または p += 1;
```

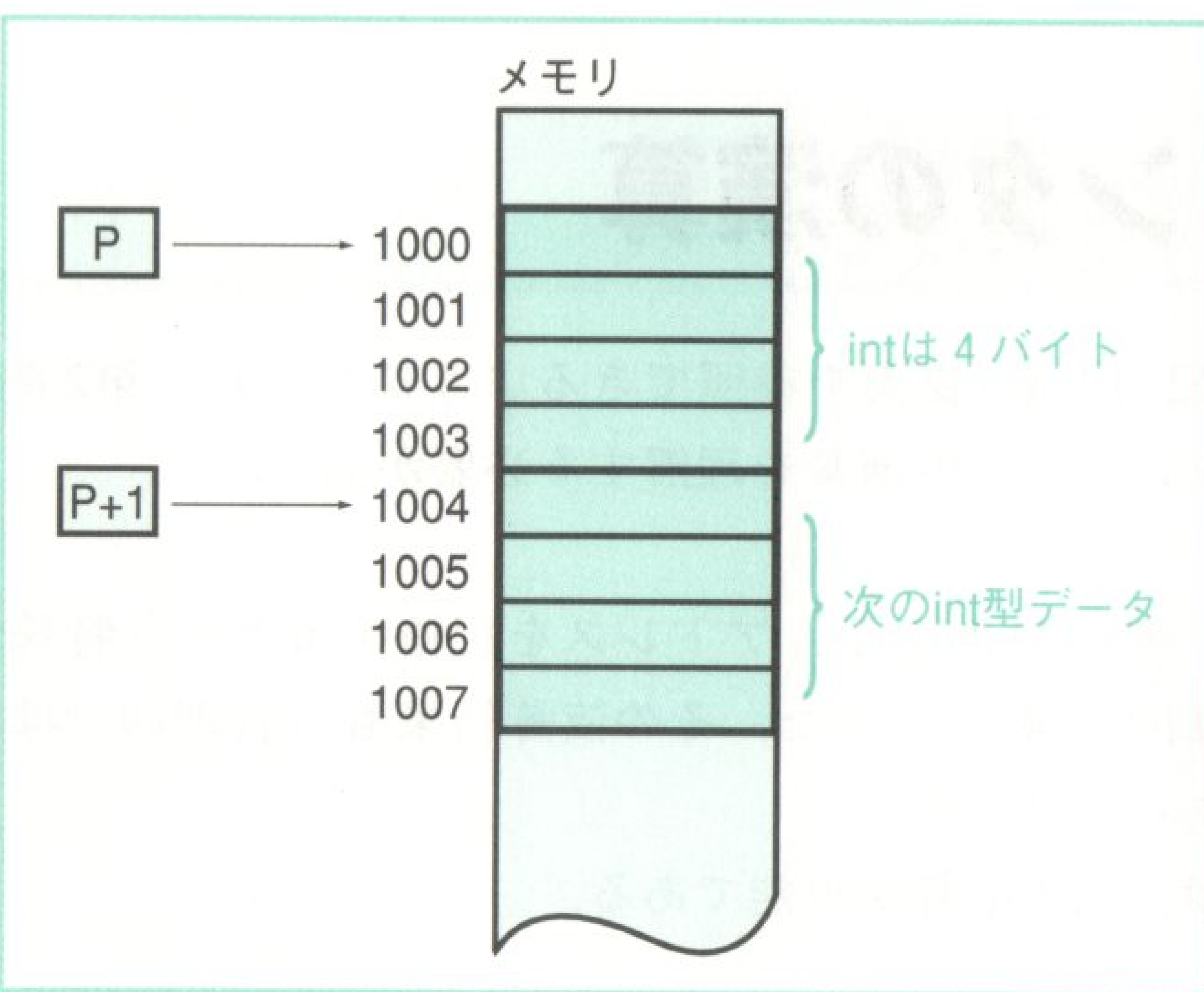
1001？――残念ながら，答えは1004である．1を加えると，1ではなく，4大きくなるのである．この4はsizeof（int）のことである．つまり，int型データが使用するメモリのサイズ分大きくなる．

一般に，ポインタにxを加算すると，sizeof（type）* x分だけ大きくなる．ここでtypeは，そのポインタの型である．

なぜ，このような変な値が加算されるのか？　それは，そのほうが便利だからである．

いまのint型のポインタpを例に説明する．一般にint型のポインタは，それが指すアドレスにint型データが入っている．したがって，1を加算したとき1001になっても，そこには有効なデータがない．4大きくなれば，次のint型データを指すことができる．

図17-2▶



このように，ポインタにxを加えると，x個分先のメモリを指すようになる．逆にxをマイナスすると，x個分前のメモリを指すようになる．

同様の考え方で，その他の演算も理解することができる．

ポインタpについての演算

整数値の加算：p + n
　　pが指しているところからn番目の要素のアドレス

整数値の減算：p - n
　　pが指しているところからn番前の要素のアドレス

アドレスとの比較
　　比較演算子<，<=，==，!=，<，<=を使って
　　アドレスの大小を比較する

ポインタ通しの減算
　　両者のアドレスの差に記憶できる要素数

3 配列要素の参照

ポインタの演算がわかれば，配列の第2要素以下を参照することができる．やり方は2つある．ポインタを移動しないで参照する方法と，ポインタを移動しながら参照する方法の2つである．

3.1◆ポインタを移動しない

ポインタを移動しない方法は，次のように行う．

p ―――――――――― 配列の第1要素のアドレス
p + 1 ―――――――― 配列の第2要素のアドレス
P + 2 ―――――――― 配列の第3要素のアドレス

したがって，これに逆参照演算子を付ければ，配列要素の値を取り出すことができる．

*p ―――― 配列の第1要素の値
*(p + 1) ―――― 配列の第2要素の値
*(P + 2) ―――― 配列の第3要素の値

このようにして配列要素を参照しても，pの値は変わらない．相変わらず配列の先頭アドレスを指している．

この方法で，各配列要素の値を出力するプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

int main()
{
    int d[] = {
        1111,
        2222,
        3333,
    };

    int *p = d;  ―――― ポインタを宣言すると同時に
                      配列名を代入

    cout << *p << endl
         << *(p + 1) << endl
         << *(p + 2) << endl;

    return 0;
}
```

【実行例】

```
% a.out
1111 ―――― これはd[0]の値
2222 ―――― これはd[1]の値
3333 ―――― これはd[2]の値
% _
```

3.2◆ポインタを移動する

ポインタを移動しながら配列の要素を参照するには，++pとかp += 1のようにして，ポインタの値を次の配列要素を指すように変更してしまう．あとは*pで次の配列要素を参照することができる．

この方法は，反復文を使って全配列要素を参照するのに向いている．

【サンプルプログラム】

```
#include <iostream.h>

int main()
{
    int d[] = {
        1111,
        2222,
        3333,
    };
    int *p;

    p = d;
    for (int i = 0; i < 3; i++)     // 反復文を使う
        cout << *p++ << endl;

    return 0;
}
```

【実行例】

```
% a.out
1111 ―――― これはd[0]の値
2222 ―――― これはd[1]の値
3333 ―――― これはd[2]の値
% _
```

このプログラムでは，次の文がポイントである．

cout << *p++ << endl;

これは，現在の配列要素を出力すると同時に，ポインタpを次の配列要素を指すようにインクリメントしている．

4 ポインタと配列の相互利用

ポインタをあたかも配列のようにして使用することができる．また，配列名をポインタのようにして使用することができる．

4.1◆ポインタを配列として使う

ここまでは，配列のアドレスをポインタにセットし，逆参照演算子を使って配列の要素を参照した．つまり，ポインタをポインタとして使用してきた．しかし，ポインタをあたかも配列のように使うこともできる．次に例を示す．

```
int d[] = {        ―――― 配列を宣言
    1111,
    2222,
    3333,
};
int *p;            ―――― ポインタを宣言

p = d;             ―――― ポインタに配列のアドレスを代入
```

ここまでは，第3.2節と同じである．しかし，次が異なる．

```
p[0]  ―――― d[0]と同じ
p[1]  ―――― d[1]と同じ
p[2]  ―――― d[2]と同じ
```

これは，驚異である．pはポインタである．にもかかわらず配列と同じように[]でインデックスを指定している．これで，もとの配列の要素を参照できるのである．

この方法によるサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

int main()
{
    int d[] = {
```

```
        1111,
        2222,
        3333,
    };
    int *p;

    p = d;
    for (int i = 0; i < 3; i++)
        cout << p[i] << endl; ―――――― ポインタpを配列のように使う

    return 0;
}
```

【実行例】

```
% a.out
1111 ―――――――――――――――― これはd[0]の値
2222 ―――――――――――――――― これはd[1]の値
3333 ―――――――――――――――― これはd[2]の値
% _
```

4.2◆配列をポインタとして使う

前の節とは逆に，配列をあたかもポインタのようにして使うこともできる．例を示す．

```
int d[] = { ―――――――― 配列を宣言
    1111,
    2222,
    3333,
};
```

ここで宣言したdは，もちろん配列である．これをポインタのように逆参照演算子を付けて使うと次のようになる．

```
*d ―――――――――――― 配列の第1要素の値
*(d + 1) ―――――――― 配列の第2要素の値
*(d + 2) ―――――――― 配列の第3要素の値
```

これが，きちんと使えるのである．それを次のサンプルプログラムで確かめる．

【サンプルプログラム】

```
#include <iostream.h>

int main()
{
    int d[] = {
        1111,
        2222,
        3333,
    };

    cout << *d << endl
         << *(d + 1) << endl
         << *(d + 2) << endl;

    return 0;
}
```

【実行例】

```
% a.out
1111 ———————— これはd[0]の値
2222 ———————— これはd[1]の値
3333 ———————— これはd[2]の値
% _
```

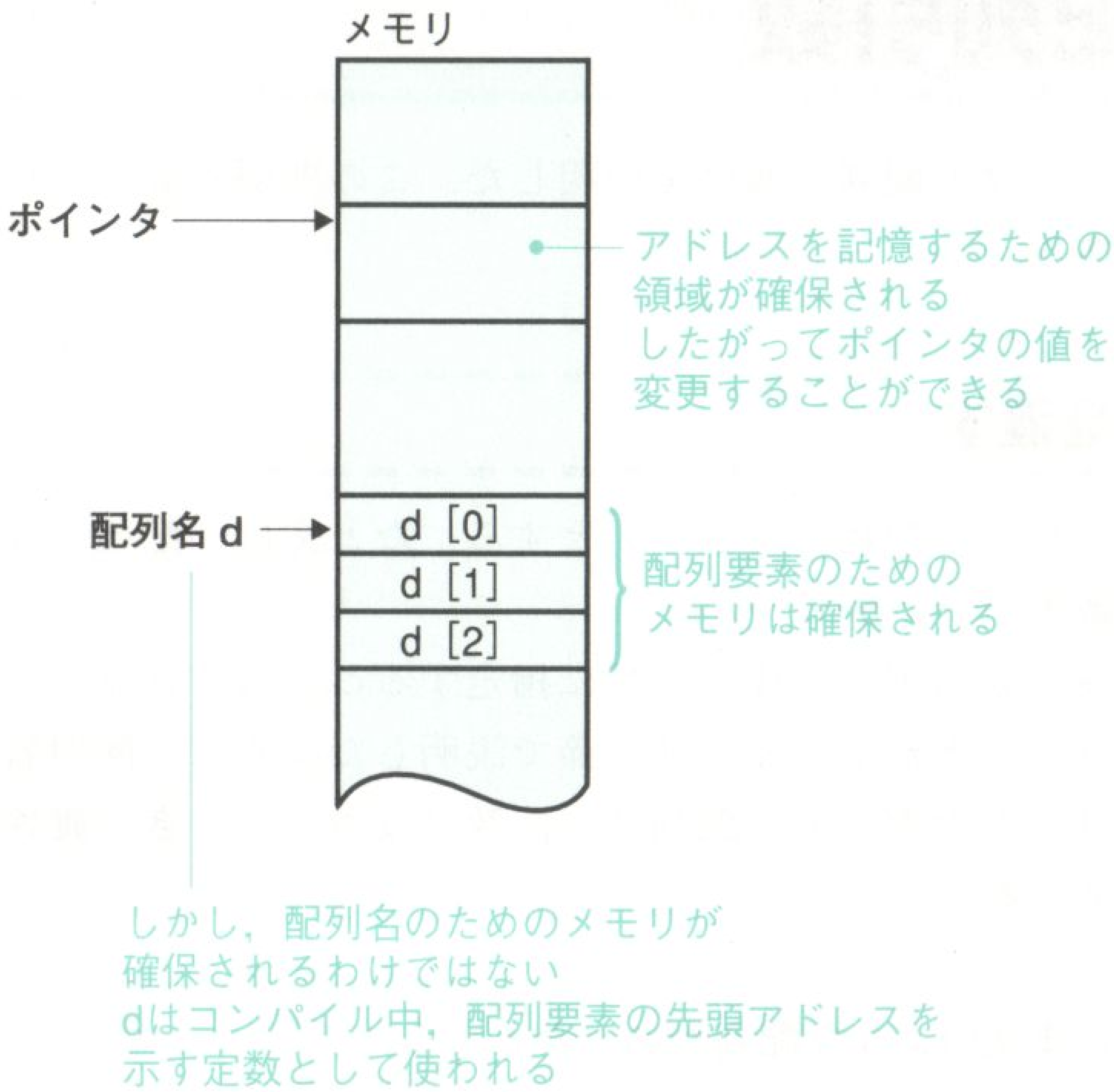
5 配列とポインタの違い

ここまでを見てくると，**ポインタ**も**配列**も変わらないように見える．しかし，両者はまったくの別物である．

ポインタはれっきとした変数なので，アドレスを記憶するための領域がメモリ上に確保される．それに対して配列の名前は，C++コンパイラがコンパイル中に使用する定数なので，領域がメモリ上に確保されることはない．その違いが大きく現れるのは次の点である．

ポインタはその値を変更できる
配列名はその値を変更できない

図17-3▶



たとえば，次のような配列とポインタがあるとする．

```
int d[3];
int p;
```

このとき，ポインタpにはアドレスを代入できるが，配列dにはアドレスを代入できない．dのためのメモリ領域がないからである．

```
d = 0x10000000;    これはダメ
p = 0x10000000;    これはOK
```

配列の名前をポインタのように使って配列要素を参照するときも，配列名を変更しようとしてはいけない．

```
*(d + 2)    これはOK

d += 2;     これはdを変更しようとしているのでダメ
*d;
```

6 配列引数

前の節でポインタと配列の関係を説明した．この関係が最もシビアに現れるのが配列引数である．

6.1◆配列を渡す

引数に配列を使った関数を書きたいとする．たとえば，配列を渡すと要素の最大値を返す関数とかのように．

しかし，残念ながら配列全体を引数に指定することはできない．代わりに配列の名前を指定することができる．前の節で説明したように，配列名は配列の先頭アドレスを示す．したがって，配列名が関数に渡されるとき，渡されるのは配列の先頭アドレスである．

例を示す．いま次のような配列があったとする．

```
int d[] = {
    76639,
    2155,
    94,
    407715,
    5526
};
```

この中の最大値を求めたいとする．配列を渡すとその最大値を計算する関数maxがあるとする．maxには，配列の名前（だけ）を指定する．配列全体を指定することはできない．

```
max(d)   ──────────── これはOK
max(d[]) ──────────── これはダメ
```

6.2◆配列形式で受け取る

配列名を受け取る関数側の定義は，「配列形式」と「ポインタ形式」の2通りの書き方がある．配列形式の場合は，次のように書く（要素の型はintとする）．

```
int max(int n[])
{
    ............
}
```

配列形式で書くとき，[] の中にサイズを指定することはできない．空のままにしておく．サイズを指定したくても，関数呼び出し側から渡されるのはアドレスだけなので，指定しても意味がないからである．

6.3◆ポインタ形式で受け取る

ポインタ形式で書くときは，次のようにする（要素の型はintとする）．

```
int max(int *n)
{
    ............
}
```

受け取るのがアドレスなので，配列形式もポインタ形式もまったく同じである．関数定義の仮引数では，配列もポインタもまったく同じ扱いである．関数内部でどちらの形式を使いたいかで使い分ければよい．

6.4◆配列のサイズ

関数で配列引数を使うとき，渡されるのは配列の先頭アドレスだけである．配列のサイズは渡されない．したがって，もし関数内部で配列のサイズが必要な場合は，工夫が必要である．

いちばん簡単なのは，引数を1つ追加して配列のサイズを渡す方法である．たとえば，先のmaxなら次のようにする．

```
int max(int n[], int size)
```

ここに配列のサイズを指定してもらう（int size を指す注記）

関数maxの呼び出しは次のようになる．

```
max(d, sizeof d / sizeof d[0]);
```

6.5◆配列の最大値を求める

配列の最大値を求める関数maxの作成例を示す．この関数は，配列のアドレスとサイズを受け取り，要素の中で最大のものを見つけて，それを返す．

```
int max(int n[], int size)
{
    int max;          ←ここに最大値を見つける
    …………
    return max;
}
```

やり方はいろいろあるが，ひとまず配列の最初の要素をmaxに入れる．

```
int max = n[0];
```

そして，2番目の要素とmaxを比べ，もし2番目の要素のほうが大ならmaxの値を入れ替える．これを最後の要素まで続ければ，maxに最大値が得られる．

```
int max(int n[], int size)
{
    int max = n[0];

    for (int i = 1; i < size; i++)
        if (n[i] > max)     ←各要素を現在までの最大値と比較
            max = n[i];

    return max;
}
```

次に完成したmaxとその使用例を示す．

【サンプルプログラム】

```
#include <iostream.h>

int max(int n[], int size)
{
    int max = n[0];
```

```
    for (int i = 1; i < size; i++)
        if (n[i] > max)
            max = n[i];

    return max;
}

int main()
{
    int d[] = {
        76639,
        2155,
        94,
        407715,
        5526
    };

    cout << "最大値 = " << max(d, sizeof d / sizeof d[0]) << endl;

    return 0;
}
```

【実行例】

```
% a.out
最大値 = 407715
% _
```

第18章

文字列

C++には文字列のための変数が存在しない．それは，文字列の特殊性から，他の変数と同じ仕組みの変数は適しないからである．そこで，代わりに文字の配列やポインタを使う．

この章では，C++における文字列変数の使い方を説明する．

1 文字列変数

C++には文字列のための変数が存在しない．その理由を考察する．

1.1◆文字列変数は存在しない

すでに文字列リテラルの書き方は知っている．"Hello, world" のように文字の列を "" で囲んだ文字列リテラルを何度も使ってきた．

```
cout << "Hello, world" << endl;
```

↑ 文字列リテラル

ところが，これまで一度も文字列のための変数を使ってこなかった．その理由は簡単で，「C++には文字列のための変数が存在しない」からである．

なぜC++には文字列のための変数が存在しないないのか？　それは，C++のもととなったCに，文字列のための変数が存在しないからである．

では，なぜCには文字列のための変数が存在しないないのか？——その本当の理由は，Cの開発者にでも聞いてみないことにはわからない．しかし，文字列が他の変数と異なる特別な性質を持っていることと関係があるかもしれない．

1.2◆文字列変数の特殊性

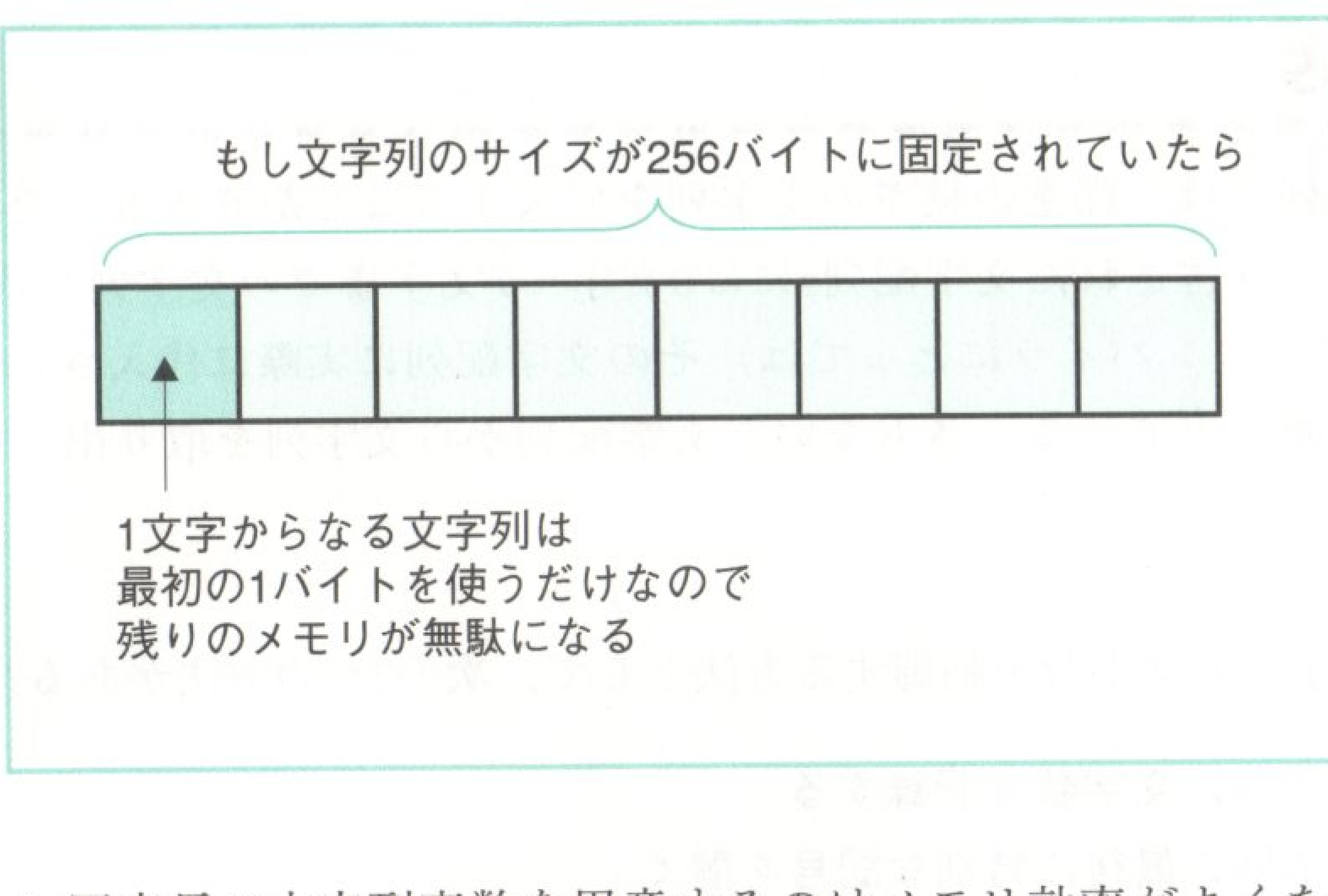
文字列の特殊性は，個々の文字列でサイズがかなり異なるということである．0文字からなる文字列もあれば，1000文字を超える長い文字列もある．このため，文字列には，他の変数のように固定長の変数を用意しにくいという特殊性がある．

たとえばchar型変数なら，どの変数でも同じ1バイトである．int型変数なら，どの変数でも同じ4バイトである．長さが固定されている（変数のサイズは処理系によって異なる）．

他の変数と同じように文字列変数を固定長にしたらどうなるか？

仮に文字列変数の長さを256バイトに固定したとする．その場合，1文字の文字列を代入すると，255バイト分のメモリが余ってしまう．逆に，その変数には256文字を超える文字列を代入することはできない．

図18-1▶



このように固定長の文字列変数を用意するのはメモリ効率がよくないし，文字列の長さに制限ができてしまう．

そこで，Cおよびそれを継承したC++では，可変長の文字列変数を使うことにした．といっても，文字列専用の変数を用意したわけではない．代わりに，すでに存在する配列を利用することにしたのである．

2 文字の配列

C++には，文字列のための変数が存在しない．代わりに文字の配列を使用する．

```
char s[10];
```
文字の配列

文字の配列を使うには，あらかじめ宣言が必要である．そのとき配列のサイズを指定する必要がある．このサイズには，代入する予定の文字列の長さ+ 1を指定するとよい．たとえば7文字の文字列を代入する予定なら次のように宣言する．

```
char s[8];
```
この配列には7文字までの文字列を代入できる

このように文字の配列を使用すれば，代入する文字列に合わせて適切なサイズの変数を利用できるのでメモリ効率がよい．

注意　文字配列を宣言するとき，文字列の長さよりも1文字分多くのサイズを確保する理由は，すぐ次のところで説明する．

2.1◆EOS

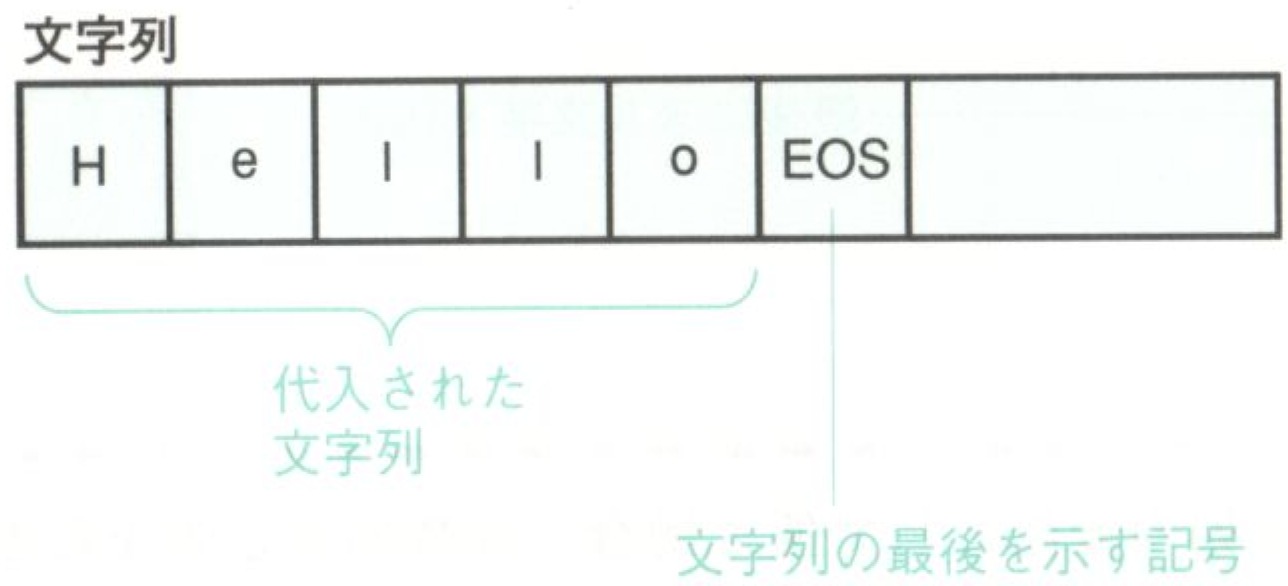
文字の配列には，任意の長さの文字列を代入することができる．たとえばchar s[8];のように宣言された文字配列sには0文字～7文字までの文字列を代入できる．すると，（C++コンパイラにとっては）その文字配列に実際に代入されている文字数を知る必要が出てくる．さもないと文字配列から文字列を取り出すことができない．

一般に文字列の文字数を制御する方法として，次の2つの方法がある．

- どこかに文字数を記録する
- 文字列の最後に特別な記号を置く

前者の方法を採用した例としてはPascal言語がある．C++は後者の方法を採用している．すなわち，文字列の最後にEOS（*End Of String*）という記号を置く．

図18-2▶



文字列の最後にはEOSという特別な文字を置く．この仕組みにより，確保した配列よりも短い文字列を代入することができるのである．

2.2◆ヌル文字

EOSの具体的な値は0（ゼロ）である．0は整数である．文字配列の各要素は文字型なので，EOSをchar型で表現したいこともある．そのときは '\0' を使う．'\0' を**ヌル文字**という．

ヌル文字
　数値による表現 ―― 0
　char型による表現 ―― '\0'

なお，ヌル文字を '0'（ゼロをアポストロフィで囲んだもの）と間違えてはいけない．

```
'0'  ―――― 文字の0（ゼロ）：値は0x30
'\n' ―――― ヌル文字列　　　：値は0x00
```

例として9文字分の文字配列を用意して，そこに“Hello”という5文字の文字列を代入するところを示す．

```
char s[10]; ―――― 文字配列の宣言

s[0] = 'H'; ―――― 文字列を1文字ずつ代入
s[1] = 'e';
```

```
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = '\0';  ――――― 最後にヌル文字を代入
```

2.3◆配列名

文字の配列を文字列の変数として使う場合，変数名が必要なときは，配列名を指定すればよい．たとえばcoutで文字列を出力するときは，配列の名前を変数として指定する．

```
cout << s << endl;  ――――― sは文字配列の配列名
```

以上をテストするためのサンプルプログラムを示す．

【サンプルプログラム】

これは，文字列変数としての文字配列の使い方を示すものである．9文字分の文字配列を用意して，そこに“Hello”という5文字の文字列を代入する．そして，coutで出力している．

```
#include <iostream.h>

int main()
{
    char s[10];  ――――――――――― 文字配列の宣言

    s[0] = 'H';  ――――――――――― 文字列の代入
    s[1] = 'e';
    s[2] = 'l';
    s[3] = 'l';
    s[4] = 'o';
    s[5] = '\0';  ――――――――――― ヌル文字

    cout << s << endl;  ――――――― coutによる出力

    return 0;
}
```

【実行例】

```
% a.out
Hello
% _
```

3 文字配列の初期化

前の節の最後に出てきたサンプルプログラムでは，「文字配列の初期化」が，かなり面倒であった．そこでは，配列要素の1つ1つに文字を代入し，最後にヌル文字を代入することで初期化していた．

実は文字配列を初期化する方法には，いろいろなやり方ある．前の節で示した方法は，その最も基本的なやり方を示したものである．しかし，（基本を別にすれば）もっと簡単な方法もある．この節では，文字配列を初期化する方法をまとめる．

3.1◆文字単位の初期化

文字配列を初期化する最も基本に忠実な方法は，前の節で示したものである．すなわち，文字列変数を文字配列と認識し，配列要素の1つ1つに文字を代入し，最後にヌル文字で締めくくる方法である．

```
char s[10]; ―――― 文字配列の宣言

s[0] = 'H'; ―――― 文字列を1文字ずつ代入
s[1] = 'e';
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = '\0'; ―――― 最後にヌル文字を代入
```

この方法については，すでに前の節で説明したので，これ以上は言及しない．

3.2◆{}を使う

文字配列を初期化する第2の方法は，宣言と同時に初期文字列を {} でくくって与える方法である．{} は，配列要素の初期化に使う書式である．文字配列の各要素はchar型なので，{} の中にはchar型のデータをカンマで区切って並べる．次に例を示す．

```
char s[] = { 'H', 'e', 'l', 'l', 'o'};
```

この方法と第3.1節の基本的な方法を比較されたい．'\0' を代入する必要がないなど，かなり簡略化できることがわかる．

1つ注意したいのは，この方法による宣言では配列のサイズを指定しないでもよいということである．

```
char s[] = { 'H', 'e', 'l', 'l', 'o'};
      ↑
サイズを指定しない
```

配列のサイズは，右辺の初期化データを収めるのにちょうどよい大きさが確保される．つまり，上記の宣言は次のプログラムと同じ意味になる．

```
char s[6]; ――――――――― "Hello" を代入するのにちょうどのサイズ

s[0] = 'H'; ――――――――― 初期化文字列を代入する
s[1] = 'e';
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = '\0'; ――――――――― 最後をヌル文字で締めくくる
```

もちろん，配列のサイズを指定してもよい．その場合は，ヌル文字を含めて初期化する文字列を保存するのにちょうどよいサイズか，それよりも大きなサイズを指定すること．

3.3◆文字列リテラルを使う

第3.2節の宣言は，だいぶ直観的になったとはいえ，初期化文字列の文字を1つずつ '' で囲まなければならないという煩わしさが残る．

文字配列のサイズを初期化文字列のサイズにしてよいなら，もっと簡便な方法がある．右辺に直接文字列リテラルを指定してしまうのである．次に例を示す．

```
char s[] = "Hello";
```

これにより6バイトの文字配列が確保され，右辺の文字列がそこに保存される．文字列の最後には，ちゃんとヌル文字が付加される．

これなら直観的に書ける．これを第3.2節の書き方と比べると，簡便なのがわかる．

```
char s[] = { 'H', 'e', 'l', 'l', 'o'};    ——— 第3.2節の書き方
```

ここでは配列のサイズを指定しなかったが，もちろん指定してもよい．その場合は，ヌル文字を含めて初期化する文字列を保存するのにちょうどよいサイズか，それよりも大きなサイズを指定すること．

3.4◆同じ文字で初期化する

もしすべての文字を同じ文字列で初期化するなら，memsetという関数が使える．この関数を使う場合は，string.hをincludeする必要がある．

```
#include <string.h>
```

memsetの書式は，次のようになっている．

```
memset(文字配列の名前,初期化する文字,文字数);
```

たとえば“AAAAAAA”('A' を7文字) で初期化するなら次のようにする．

```
char s[8];
memset(s, 'A', 7);
```

memsetはヌル文字を付加しないので，手動でヌル文字を追加する必要がある．

```
s[7] = '\0';
```

【サンプルプログラム】

これは，文字配列を同じ文字で初期化する例である．結果の文字列を出力した後，個々の配列要素を16進数のASCIIコードで出力する．

```
#include <iostream.h>
#include <string.h>

int main()
{
    char s[8];                          文字配列を確保

    memset(s, 'A', 7);                  "A"で初期化
    s[7] = '\0';                        ヌル文字の付加

    cout << s << endl;
    for (int i = 0; i < 8; i++)
        cout << hex << (int) s[i] << endl;   ASCIIコードの出力

    return 0;
}
```

【実行例】

```
% a.out
AAAAAAA          "A"が7文字
41               s[0]
41               s[1]
41
41               s[3]
41
41
41               s[6]
0                s[7]
% _
```

3.5◆文字列の代入

文字配列を初期化する方法を説明してきたが，任意の時点で文字配列に文字列を代入するときは注意が必要である．次のように代入演算子=を使って代入することができないからである．

```
s = "new string";    これはダメ
```

sは，文字配列の先頭アドレスを示す定数なので，このような代入はできない．

あくまでも文字配列として，各要素に1文字ずつ代入する必要がある．

```
s[0] = 'n';
s[1] = 'e';
s[2] = 'w';
s[3] = ' ';      この右辺はスペース文字である
s[4] = 's';
s[5] = 't';
s[6] = 'r';
s[7] = 'i';
s[8] = 'n';
s[9] = 'g';
s[10] = '\0';    やはりヌル文字が必要
```

さらに重要な注意として，文字配列sは新しい文字列を代入するのに十分な大きさを持っていなければならない．たとえば上のような代入を行うには，sに11バイト以上のサイズが必要である．C++コンパイラは，代入において配列のサイズを考慮しないので，不正な代入をしてもエラーを報告しない．もし，そのような代入を行うと，他の変数の値を壊してしまうだろう．あるいは，もっと悲惨なことになるかもしれない．

文字列の代入

- 配列要素に1文字ずつ代入する
- 文字列の最後にヌル文字を加える
- 文字配列のサイズが十分であること

文字列を代入するのに，1文字代入するのは面倒である．そこで，strcpyという専用の関数が用意されている．strcpyについては，第6.2節で説明する．

4 文字へのポインタ

前の節まで，文字列変数として文字の配列を使ってきた．代わりに**文字へのポインタ**を使う方法もある．

```
char *s;
```

この節では，ポインタを使った文字列変数を説明する．

4.1◆メモリの確保

ポインタを使う場合は，ポインタを宣言しただけでは文字列のためのメモリが確保されていないことに注意．

```
char *s; ―――――――――――――――― (1)
```

すなわち（1）の宣言だけでは，文字列を保存するためのメモリがまだ存在しない．文字列のためのメモリを確保するには，次のような方法がある．

- 第2部で説明するフリーストアを利用する
- 文字配列を確保して，その先頭アドレスをポインタに代入する

後者の方法では，次のような感じになる．

```
char *p; ――――――――――――――――――― ポインタを宣言
char s[] = "ABCDEFGHIJ"; ――――― 文字列用のメモリを確保
p = s; ――――――――――――――――――――― そのアドレスをポインタに代入
```

こうすれば，文字列をポインタを通して操作できる．たとえばp++でpを2文字目を指すようにインクリメントすれば，あたかも1文字目が消えたように見える．しかし，もとの文字列はそのまま残っているので，p--でもとの文字列に戻る．

```
cout << p; ――――――――――――――― "ABCDEFGHIJ" が出力される
p++; ―――――――――――――――――――― pは2文字目を指す
cout << p; ――――――――――――――― "BCDEFGHIJ" が出力される
p--; ―――――――――――――――――――― pは2文字目を指す
cout << p; ――――――――――――――― "ABCDEFGHIJ" が出力される
```

4.2◆文字遊び

ポインタを使った文字列の例として「文字遊び」を紹介する．

【サンプルプログラム】

このプログラムは，文字列を面白く表現するものである．最初に実行例を見て

いただき，その後でプログラムを説明する．

```
#include <iostream.h>

int main()
{
    char *p;
    char s[] = "ABCDEFGHIJ";

    p = s;

    while (*p)                       文字列を減らしていく
        cout << p++ << endl;
    while (*p != 'A')
        cout << p-- << endl;         文字列を増やしていく
    cout << p << endl;

    return 0;
}
```

【実行例】

```
% a.out
ABCDEFGHIJ      "ABCDEFGHIJ" が1文字消えていく
BCDEFGHIJ
CDEFGHIJ
DEFGHIJ
EFGHIJ
FGHIJ
GHIJ
HIJ
IJ
J
                とうとう文字数0になる
J               1文字復活する
IJ
HIJ
GHIJ
FGHIJ
EFGHIJ
DEFGHIJ
CDEFGHIJ
BCDEFGHIJ
ABCDEFGHIJ      もとの文字列に戻る
% _
```

実行例を見ればわかるが，ポインタpが指す文字列が1文字ずつ減っていき，空文字列になった後，1文字ずつ復活していく．

文字列を減らす部分は，p++でpが次の文字を指すようにすることで実現している．

```
cout << p++ << endl;
```

現在の文字列を表示した後，
次の文字を指すようにインクリメント

そのとき，次のwhile文を使っている．

```
while (*p)
    cout << p++ << endl;
```

この条件式（*p）がわかりづらいかもしれない．これは，次の条件式を略したものである．

```
while (*p != '\0')
```

*pは，pが指す文字列の先頭文字である．したがって，この条件式はヌル文字になるまで繰り返すという意味になる．

ヌル文字の実体は0なので，（*p != '\0'）は（*p != 0）と同じである．!= 0はなくてもよい．ゼロでなければ真になるからである．つまり，次の3つは同じことをいっている．

```
(*p != '\0')
   ↓
(*p != 0)
   ↓
(*p)
```

文字列を増やす方も，同じ考え方で実現できる．

4.3◆ポインタ文字列の初期化

文字列変数としてポインタを利用する場合，ポインタとは別に文字列のためのメモリを確保する必要がある．しかし，ポインタを宣言すると同時にメモリを確保する方法がある．

```
char *p = "Hello, world";
```

この書式とC++の振る舞いは面白い．まず右辺の文字列が文字列リテラルとしてメモリ上に確保される．その先頭アドレスをポインタpが指す．

この書式は書きやすいので，第3.3節の方法と並んでよく使われる．ただし，この方法による制限は，確保した文字列を変更することができないということである．たとえば，次のようなことはできない．

```
p[0] = 'h';   ―――― 1文字目を小文字のhに変更しようとした
```

このプログラムを実行すると，coreを吐いて停止する．

【サンプルプログラム】

これは，文字列ポインタを宣言すると同時に初期文字列を確保した例である．

```
#include <iostream.h>

int main()
{
    char *p = "Hello, world";   ―――― 宣言と同時に文字列の確保
    cout << p << endl;

    return 0;
}
```

【実行例】

```
% a.out
Hello, world
% _
```

5 文字列引数と文字列関数値

この節では，文字列を引数にとる関数，および文字列を関数値として返す関数の書き方を説明する．

5.1◆文字列引数

文字列を引数にとる関数を作るには，第17章第6節「**配列引数**」での説明がそのまま適用される．すなわち，関数定義の仮引数は，次のどちらの書き方をしてもよい．

```
void func(char s[])    ―― 配列形式
void func(char *s)     ―― ポインタ形式
```

配列形式で書くときは配列のサイズを指定しないことに注意．

この関数を呼び出すときは，文字列の先頭アドレスを渡す．たとえば文字配列の場合は，配列の名前を指定する．

```
char str[] = "Computer";
func(str);     ―― 配列の名前を指定する
```

文字ポインタの場合は，そのポインタを指定する．

```
char *p = "Computer";
func(p);     ―― ポインタを指定する
```

次のように文字列リテラルを直接指定することもできる．

```
func("Computer");
```

この場合も文字列の先頭アドレスが関数に渡される．というのは，文字列リテラルが式の中で使われると，その先頭アドレスとして処理されるからである．

5.2◆文字列関数値

関数値として文字列を返す場合は，文字のポインタを使う．

```
char *func()     ―― この関数は文字列を返す
```

この関数を使うときは，次のようにして文字列を受け取る．

```
char s[256];     ―― 関数が返す文字列を保存するための配列
s = func();
```

次の項で，文字列を受け取り文字列を返す関数の例を示す．

5.3◆最後の文字を削除する

次のプログラムは，declastという関数の作り方を示すものである．declastは，引数として文字列を受け取り，その最後の文字を削除して返す．

【サンプルプログラム】

```
#include <iostream.h>
#include <string.h>      // for strlen

// 最後の文字を取り除く
char *declast(char *s)
{
     int last = strlen(s);
     if (last > 0)
          s[last - 1] = '\0';

     return s;
}

int main()
{
     char s[] = "Computer ";

     while (*s) ――――――――――――1文字以上の長さがあれば
          cout << declast(s) << endl; ――――1文字削除する

     return 0;
}
```

【実行例】

```
% a.out
Computer ――――――――――――文字列が1文字ずつ削除されていく
Compute
Comput
Compu
Comp
Com
Co
C

% _
```

このプログラムは，文字列の長さを調べるのにstrlenという関数を使用している．この関数を使うには，次のファイルをincludeする必要がある．

```
#include <string.h>
```

strlenを使って長さを調べた理由は，渡された文字列が1文字削除するのに十分な長さがあるかを調べるためである．

```
if (last > 0)            ——1文字以上の長さがあれば
    s[last - 1] = '\0';  ——1文字削除する
```

6 文字列関数

C++には文字列のための変数が存在しない．そのため，文字列変数として文字配列や文字ポインタを使うわけであるが，その代償として文字列変数の間で演算を行うことができないという制限が加わる．たとえば，文字列変数と文字列変数を+して文字列を連結しようとしても，それはできない．

```
char *p1 = "Hello";
char *p2 = ", world";
p2 += p1;    ——"Hello, world" になればよいのだが
                そうはならない
```

そこで，文字列に対する関数がいろいろ用意されている．以下，有用な文字列関数の使い方を説明していく．

6.1◆数値への変換

ここで紹介する関数は，文字列を数値に変換する．

```
int atoi(const char *nptr);
                  ——文字列をintに変換する

long atol(const char *nptr);
                  ——文字列をlongに変換する
```

```
double atof(const char *nptr);
            ──────────────────文字列をdoubleに変換する
```

数値を示す文字列の前に+や-の記号があってもよい．次に例を示す．

```
int n = atoi("8304601");
            ──────────────────n = 8304601になる

long l = atol("-120000000");
            ──────────────────l = -120000000になる
```

atofは文字列を浮動小数点数に変換するが，その型はfloatではなく，doubleであることに注意．

```
double d = atof("1.2345e2");
            ──────────────────d = 123.45になる
```

もし，数値に変換できない文字があるときは，その文字の手前までを変換する．ただし，先頭の空白文字は無視される．また，まったく数値に変換できないときは0に変換する．

```
int n = atoi("      63419Happy!");
            ──────────────────n = 63419になる

long l = atol("Hello");
            ──────────────────l = 0になる
```

これらの関数を使うときは，次のファイルをincludeすること．

```
#include <stdlib.h>
```

6.2◆コピー

文字列を他の文字列変数にコピーするには次の関数を使う．

```
char *strcpy(char *dest, const char *src);
            ──────────────────srcからdestにコピーする

char *strncpy(char *dest, const char *src, size_t n);
            ──────────────────srcの先頭からdestにn文字コピーする
```

次にstrcpyの使用例を示す．

```
char dest[100];
            ―――――受け側の文字列領域を確保する
char src[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
            ―――――送り側の文字列を用意する
strcpy(dest, src);
            ―――――dest = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" になる
```

strncpyで注意しなければならないのは，コピーした文字列の最後にヌル文字が付加されないということである．したがって，自分でヌル文字を書き込む必要がある．

次にstrncpyの使用例を示す．

```
char dest[100];
            ―――――――――受け側の文字列領域を確保する
char src[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
            ―――――――――送り側の文字列を用意する
strncpy(dest, src, 10);
            ―――――――――"ABCDEFGHIJ" がコピーされる
                          ただし，ヌル文字が付加されないので
                          このままではdestを使えない
dest[10] = '\0';
            ―――――――――ヌル文字を付加
                          dest = "ABCDEFGHIJ" になる
```

strncpyでsrcの長さがn文字よりも短いときは，残りの文字としてヌル文字がコピーされる．したがって，この場合は，srcがそっくりコピーされたような感じになり，自分でヌル文字を付加する必要はない．

これらの関数を使うときは，次のファイルをincludeする．

```
#include <string.h>
```

6.3◆連結

文字列と文字列を連結して1つの文字列にするには次の関数を使う．

char *strcat(char *dest, const char *src);

strcatはdestの後ろにsrcを連結する．このため，descは連結された文字列を保存するのに十分なサイズを持っていなければならない．

次に例を示す．

char s1[20] = "ABCDEFGHIJKLMN";
――連結できるサイズを確保

char s2[] = "12345678";

strcat(s1, s2);
――s1 = "ABCDEFGHIJKLMN12345678" になる

この関数を使うときは，次のファイルをincludeすること．

#include <string.h>

6.4◆文字列の検索

次の関数は，文字列s1の中に文字列s2が含まれているかを検索する．

char *strstr(const char *s1, const char *s2);

見つかった場合は，その位置を示すポインタを返す．見つからない場合は，0の値を持つポインタを返す．

この関数を使うときは，次のファイルをincludeすること．

#include <string.h>

【サンプルプログラム】

次は，関数strstrの使い方を示すためのプログラムである．“Hello, world”の中から“wo”を検索する．

```
#include <iostream.h>
#include <string.h>

int main()
{
    char *s = "Hello, world";
    cout << strstr(s, "wo") << endl;
```

```
    return 0;
}
```

【実行例】

```
% a.out
world ―――――― 見つかった位置から残りの文字列を返す
% -
```

オブジェクト指向
プログラミング

PART
2

第19章 クラス

C++によるオブジェクト指向プログラミングでは，**クラス**を単位にプログラムを構築していく．したがって，クラスを理解することがオブジェクト指向プログラミングの入口になる．

この章では，クラスを定義し，クラス型の変数を宣言し，それを使うまでの一連の流れを説明する．ただし，この章で定義するクラスには次の制限が残る．

- メンバー関数の欠如
- コンパイル不可

これらの制限は，次の章で解除される．

1 クラスの実態

C++によるオブジェクト指向プログラミングに入る前に「クラス」がどのようなものか，その本質を掴(つか)んでおく．

1.1◆ユーザー定義型

第15章「配列」でも説明したように，C++には2つの種類のデータ型がある．

基本型 ―― C++に最初から組み込まれているデータ型
char，int，floatなど

ユーザー定義型 ―― 必要に応じてユーザーが定義した型

クラスというと**オブジェクト指向**の匂いがプンプンして難しそうに聞こえるが，要するに，ユーザー定義型の1つに過ぎない．たとえばCustomerというクラスを作ったとする．Customerは，intとかcharとかと同じように型名の1つに過ぎない．したがって，次のようにして変数宣言をすることができる．



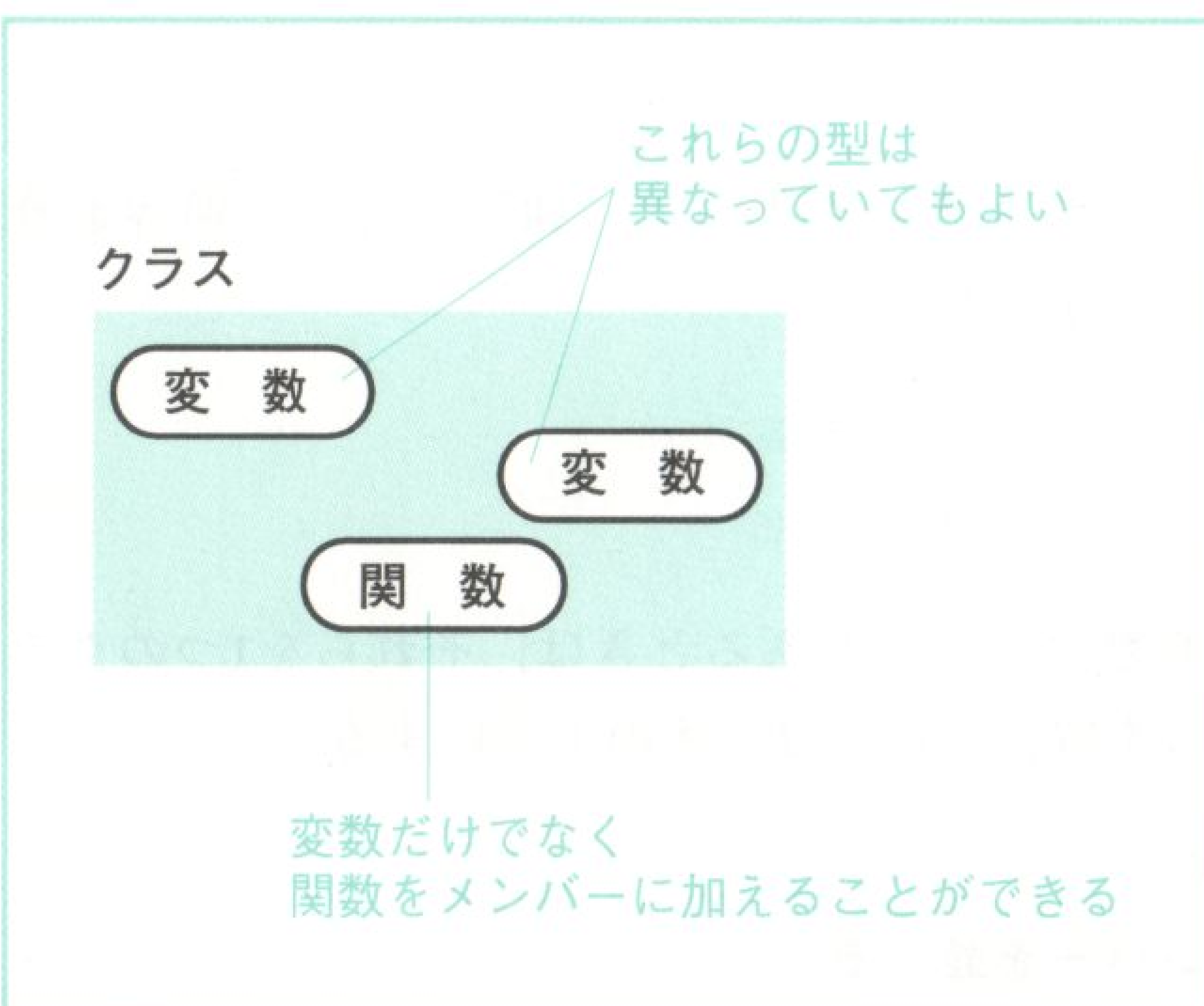
あとは，このsomeoneの使い方を覚えれば，クラス（という型のデータ）を使うことができる．なぜなら，データ型によって変数の使い方は異なる．クラス型変数の使い方は，他の基本型変数の使い方とは異なる．ゆえに，クラス型変数の使い方を覚えるのが，C++によるオブジェクト指向プログラミングの入口になる．

1.2◆クラスのメンバー

クラスは，データ型の1つに過ぎない．恐れることはない．では，クラスとはどのようなユーザー定義型だろう？

配列は同じ型のデータを集めた集合であった．それに対してクラスは，異なる型のデータを集めた集合である．それどころかクラスは，関数さえもメンバーにすることができる．

図19-1▶



このように，クラスは変数と関数をメンバーにすることができる．変数のメンバーを**メンバー変数**という．また，関数のメンバーを**メンバー関数**という．

クラスのメンバー

メンバー変数
メンバー関数

注意　言語によっては，メンバーの関数や手続きをメソッドということもある．

次の節からクラスの作成を始める．最初は関数を持たないクラスから始め，除々に高度なクラスに進んでいく．

2 クラス宣言

これから新しいクラスを宣言する．C++では，プログラムに新しい名前を導入することを**宣言**という．「名前を導入する」というのは，その名前をプログラムで使えるようにすることである．その名前が何であるかをC++コンパイラに認識させることである．

クラスを作成するということは，新しいデータの名前を導入することなので，宣言である．

いま顧客データを扱うプログラムがあって，顧客ごとに「顧客番号」と「顧客名」を管理しなければならないとする．

```
int id; ―――――――――― 顧客番号
char name[20]; ―――――― 顧客名
```

このように関連する複数のデータがあるときは，それらを1つのクラスにまとめることができる．クラスを宣言するには，次のようにする．

```
class クラス名 {
    // ここにメンバーを並べる
};
```

クラス名が新しいデータ型の名前になる．顧客データ用のクラス名をCustomerとすれば，クラス宣言は次のようになる．最後の } の後ろにセミコロンを付けるのを忘れないように！

```
class Customer {
    ......
};
```

注意　クラス名は，他の名前（たとえば変数名）と同じ規則で付けることができる．どのような名前を付けるかは自由である．本書では，クラス名の先頭を大文字で始めることにする．

クラスCustomerには，2つのメンバーを入れる．

```
class Customer {
    int id;  ――――――――― 顧客番号
    char name[20]; ――――― 顧客名
};
```

これで，idやnameがクラスCustomerのメンバー変数になる．

3 クラスの利用

classで宣言されたクラスは，新しい**型**を宣言しただけであって（型名をプログラムに導入しただけであって），それだけでそのような変数が作られるわけではない．クラス型のデータを使うには，他の変数と同様に，変数宣言が必要である．

3.1◆クラス型変数の宣言

クラス型の変数を使うときも，（一般の変数のように）変数宣言が必要である．宣言のしかたは，基本型変数の宣言と同じである．

型　変数名;

たとえばCustomerというクラス型の変数を宣言するなら次のようにする．

```
Customer someone;
```

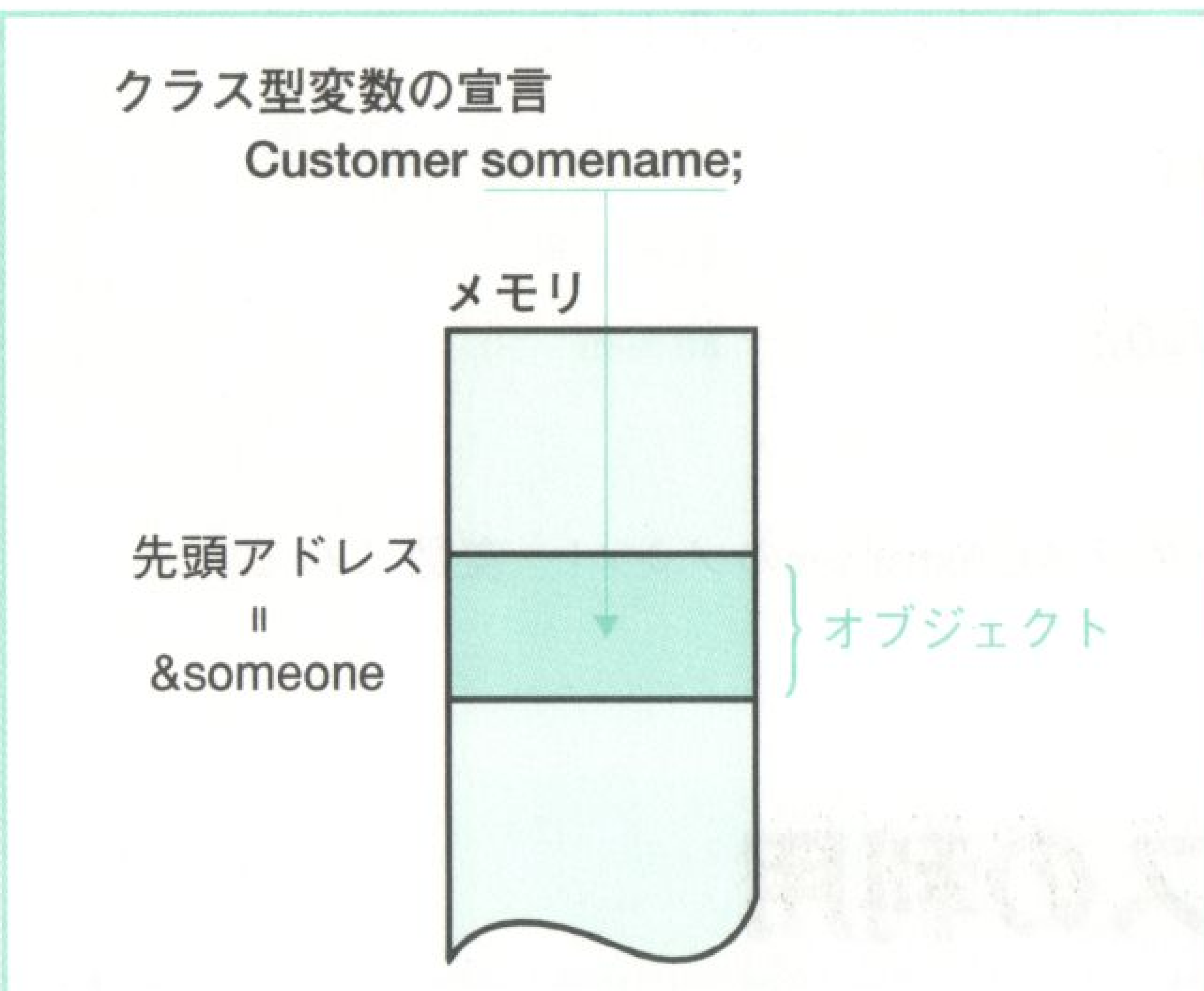
これで，someoneというCustomer型の変数を使うことができる．

3.2◆オブジェクト

クラス型の実体――すなわち，クラス型データを実際にメモリに割り当てたものを**オブジェクト**または**インスタンス**という．

クラス型の変数を宣言すると，その型のデータがメモリに割り当てられる．つまり，オブジェクトが作成される．変数名は，そのオブジェクトの名前である．また，変数名の前に&を付けたものが，そのオブジェクトが割り当てられたメモリの先頭アドレスを意味する．

図19-2▶



これからは，クラス型変数を**オブジェクト**と呼ぶことにする．

3.3◆宣言と定義

ここで，**宣言**と**定義**という用語について，その違いを明らかにしておく．

C++では，プログラムに名前を導入することを**宣言**（*declaration*）という．クラスの宣言というのは，クラス名という名前を導入するので，宣言である．

一方，**定義**（*definition*）というのは，その名前の実体を定義することをいう．具体的には，次のようなことを指す．

- 変数ならメモリを確保する
- 関数ならその本体を定義する
- 定数（変数の前にconstを付ける）ならその値を定義する

```
const int PI = 3.14159;
```

- typeによる新しい型名を導入することも定義である

```
typedef newname sometype
```

たとえば，関数の定義は，定義である．関数を定義すると，プログラムの実行時にその関数のためのメモリが確保され，そこに関数のコードが読み込まれる．しかし，関数のプロトタイプ宣言は，定義ではない．実際の定義は，どこか別の場所にある．

変数宣言は，宣言であり，かつ定義でもある．なぜなら，変数を宣言すると変数名という名前がプログラムに導入される．ゆえに宣言である．また，プログラムの実行時にその変数のためのメモリが確保され，そこに値を保存することができる．ゆえに定義でもある．

変数宣言のように宣言と定義が同時に行われるのを**定義宣言**という．

4 メンバー変数の参照

Customer型の変数someoneは，idとnameの2つのメンバー変数を持っている．

```
class Customer
    int id; ―――――― 2つのメンバー変数
    char name[20];
};

Customer someone; ――― someoneはidとnameのメンバーを持つ
```

これらのメンバー変数に直接値を代入したり，値を参照したりすることはできない．

```
id = 100; ―――――― これはダメ
cout << name << endl; ―― これもダメ
```

なぜか？　それは，どの顧客についてのメンバー変数なのかわからないからである．たとえば，次のように2人の顧客c1，c2がいたとする．

```
Customer c1, c2;
```

当然ながら，顧客c1と顧客c2のid（顧客番号）は異なる．また，name（顧客名）も異なる．単純にid = 100;とやっては，c1さんのidをセットするのか，c2さんのidをセットするのかわからない．

クラスのメンバーを参照するには，次のようにメンバーの前にクラス名を付ける．クラス名とメンバーの間をピリオドで区切る．こうすれば，どのオブジェクトのメンバーを参照するのかを明らかにする．

```
c1.id = 100; ―――――― c1のidをセットする
c2.id = 100; ―――――― c2のidをセットする
```

5 顧客クラス

次にクラスの宣言からメンバー変数を参照するまでの流れをサンプルプログラムにまとめたものを示す．

【サンプルプログラム】

これは，Customerというクラスを宣言し，cというクラス型変数を宣言し，メンバー変数を参照するまでの流れを示したものである．ただし，このプログラムは，まだ完全ではない．

```
#include <iostream.h>

class Customer { ―――――――――― クラスの宣言
    int id;
    char name[20];
};

int main()
{
    Customer c; ―――――――――― クラス型変数の宣言
```

```
    c.id = 100;                     ──── メンバー変数への代入
    strcpy(c.name, "Stallman");
    cout << c.id << endl            ──── メンバー変数の参照
         << c.name << endl;

   return 0;
}
```

残念ながら，このプログラムは，このままではコンパイルできない．コンパイルしようとすると，次のようなエラーになる．

【コンパイル例】

```
% g++ go.C
go.C: In function `int main()':
go.C:13: member `id' is a private member of class `Customer'
go.C:14: member `name' is a private member of class `Customer'
go.C:15: member `id' is a private member of class `Customer'
go.C:16: member `name' is a private member of class `Customer'
% _
```

このエラーを取るには，次の章で説明するアクセス制御キーワードを知る必要がある．

第20章 アクセス制御

クラスの設計を変更しても，プログラム全体に及ぼす影響を最小にするためには，参照してよいメンバーと参照してはいけないメンバーを分けるのがよい．そのための仕掛けとして，**アクセス制御キーワード**が用意されている．

この章では，アクセス制御キーワードについて説明する．これにより，クラスについての基礎が一段落する．

1 顧客クラス（第2版）

前の章の最後のところで示したサンプルプログラムをコンパイルできるようにするには，アクセス制御を行うためのキーワード**public**を追加する必要がある．

```
class Customer {
public:   ← ここ
    int id;
    char name[20];
}
```

この1行を追加するだけでコンパイルできるようになる．publicがどのような意味を持つのかは，まだ説明しない（第4節で説明する）．ここでは，「publicを追加することにより，クラスの外からメンバー変数を参照できるようになる」とだけいっておく．

次に前の章のサンプルプログラムをコンパイルできるように修正したものを示す．

【サンプルプログラム】

```
#include <iostream.h>

class Customer {
public:                         アクセス制御キーワードを追加した
    int id;
    char name[20];
};

int main()
{
    Customer c;

    c.id = 100;
    strcpy(c.name, "Stallman");
    cout << c.id << endl
         << c.name << endl;

    return 0;
}
```

今度は，エラーが発生することなく，コンパイルすることができる．次に実行例を示す．

【実行例】

```
% a.out
100                  顧客番号
Stallman             顧客名
% _
```

2 拡張に強いクラス

プログラムを利用していくうちに，業務が拡大するなどしてクラス内部のデータ構造を変更する必要が出てくることがある．メンバー変数の型を変更したり，新しいメンバー変数を追加したりといったように．

そのようなときは，プログラムを書き換える必要がある．その「書き換えの手間を最小で済ませ，安全に行うには，クラスをどのように設計すればよいか？」を探るのが，この節のテーマである．

2.1◆クラスの変更が及ぼす影響

次は，第19章で紹介した顧客データ用のクラスである．

```
class Customer {
public:
    int id;             ―――― 顧客番号
    char name[20];      ―――― 顧客名
}
```

この例では，顧客番号をint型のメンバー変数idで表現している．ところが，何らかの都合で変数名のidをnumberという名前に変更しなければならないとする．

```
class Customer {
public:
    int number;         ―――― idをnumberに変更した
    char name[20];
}
```

この変更では，データ構造そのものは変化しない．きわめて単純な変更である．にもかかわらず，クラスに変更を加えると，そのクラスを利用している部分も変更しなければならない．たとえば次のように．

```
c.id        ―――― 変更前はidという名前で参照していた
 ↓
c.number    ―――― 変更後はnumberという名前で参照
```

このような変更を，プログラムの全体にわたって，漏れのないように行わなければならない．これは，かなり厳しい．つまり，クラスを変更するということは，かなり面倒なことであることがわかる．

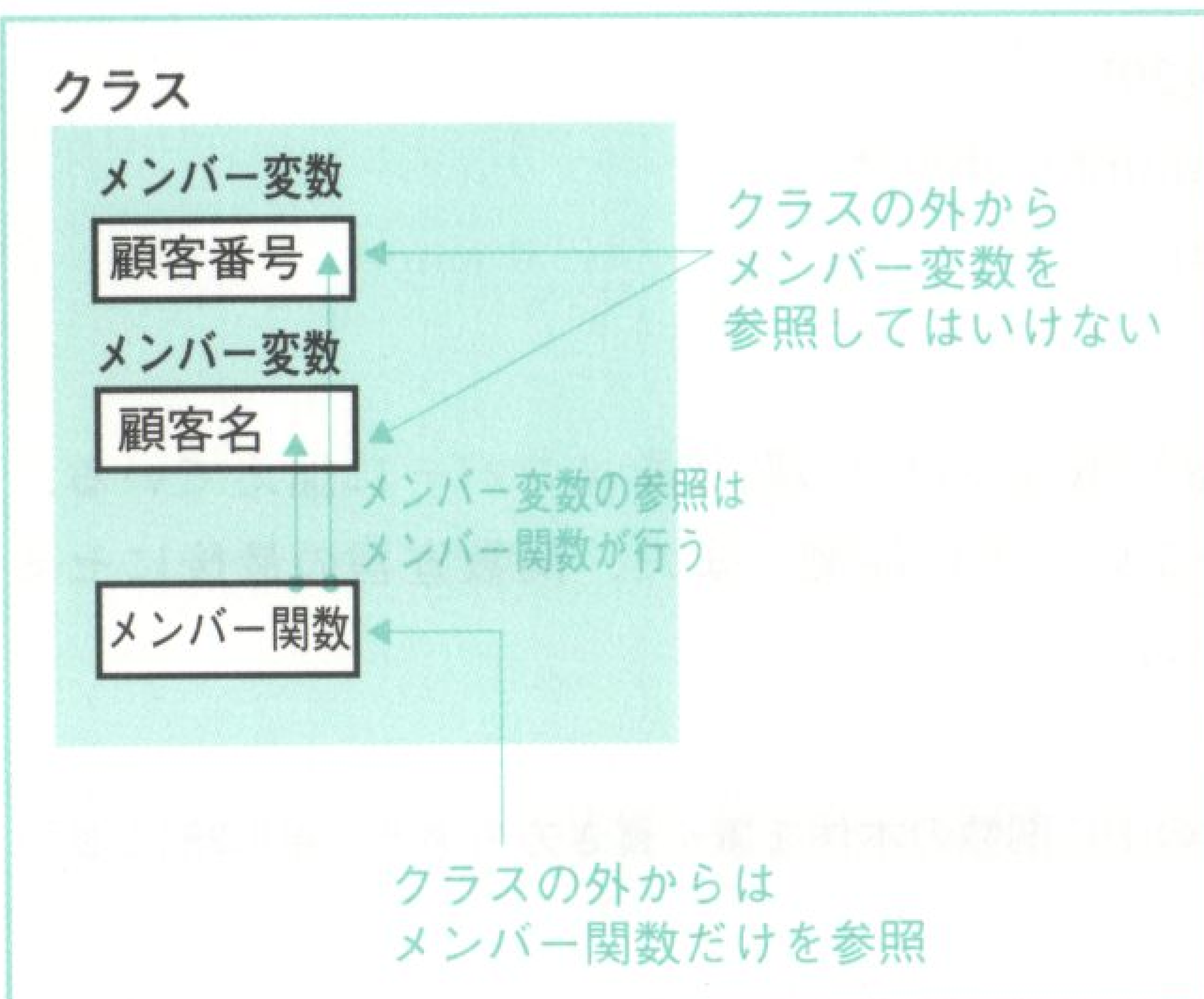
2.2◆拡張に強いクラス

クラスの作り方を工夫すると，**拡張や変更に強いクラス**を作ることができる．

クラスのメンバー変数を変更すると，そのメンバー変数を参照している部分をすべて変更しなければならない．そこで，メンバー変数を参照することを止めてしまう．こうすれば，クラス内部の変更がクラスの外部に影響しないで済むようになる．

そのためには，クラスの外からメンバー変数を参照しないでもいいような仕組みを作らなければならない．それが**メンバー関数**である．つまり，直接メンバー変数を参照する代わりに，メンバー変数を呼び出すようにするのである．

図20-1▶



たとえクラスの内部構造が変わっても，メンバー関数の仕様は変えない．こうすれば，メンバー関数を呼び出している部分は変更する必要がなく，いくらでもメンバー変数を変更することができる．データ構造を配列に変更しようが，名前を変えようが，型を変えようが自由である．

メンバー関数は，変更・拡張に強いクラスを作るときのキーになる．次の節で，クラスにメンバー関数を追加する方法を説明する．

3 メンバー関数

クラスには，変数だけでなく，関数をメンバーにすることもできる．この節では，クラスにメンバー関数を追加する方法を説明する．

3.1◆メンバー関数の追加

メンバー関数を用意すれば，クラスの外から内部の変数を守ることができる．クラスにメンバー関数を追加するには，クラス宣言の中に，その関数のヘッダ部を置けばよい．クラス宣言の中には，関数の本体を置かない．次に例を示す．

```
class Customer {
public:
    int id;
    char name[20];
    void setdata(int i, char *n); ――― メンバー関数
    void show(); ――― これもメンバー関数
};
```

この例では，setdataとshowの2つの関数をメンバーに加えている．これらの関数には，関数の本体がないことに注意．また，関数宣言の最後にセミコロンを付けるのを忘れないように．

注意　クラス宣言の中に関数の本体を置く書き方もある．第5.2節を参照されたい．

3.2◆メンバー関数の本体

メンバー関数の本体は，クラスの外で定義する．

```
class Customer {
public:
    ............
    void show(); ――― クラス内部にはヘッダだけ
};

// 悪い見本
void show()
{ ――― 本体の定義はクラスの外で
    ............
}
```

ただし，この書き方では，関数showが普通の関数なのか，クラスのメンバー関数なのか区別がつかない．つまり，この書き方では，メンバー関数ではなく，一般的な関数として扱われる．

メンバー関数を定義するには，関数名の前にクラスの名前を :: で区切って付ける．

クラス名で修飾する

```
void Customer::show()
{
    …………
}
```

これで，関数showが普通の関数ではなく，クラスCustomerのメンバー関数として扱われる．

3.3◆メンバー変数の参照

それでは，具体的にメンバー関数を定義する．クラスCustomerには，2つのメンバー関数がある．

```
class Customer {
public:
    int id;
    char name[20];
    void setdata(int i, char *n);  ←―― これと
    void show();  ←―― これ
};
```

これら2つのメンバー関数は，前者がメンバー変数の値を設定し，後者がそれを読み出す．

setdata ―― メンバー変数に値を設定する
show ―― メンバー変数の値を表示する

この2つのメンバー関数を用意することにより，クラスの外からメンバー変数を直接参照する必要がなくなる．

●setdata

このメンバー関数は，2つのメンバー変数に値を設定する．したがって，引数が2つ必要である．

```
void Customer::setdata(int i, char *n)
{
      // メンバー変数idに引数iの値を代入する
      // メンバー変数nameに引数nの値を代入する
}
```

ここで重要なことがある．メンバー変数の参照である．これまでは，メンバー変数を参照するとき，メンバー変数の前にピリオドで区切ってオブジェクト名を付けていた．しかし，ここではそれができない．クラスの中なので，オブジェクト名が存在しないからである．

このように，メンバー関数の中からメンバー変数を参照するときは，前にオブジェクト名を付けなくてよい．また，付けたくてもできない．ただ，変数名だけを書けばよい．

```
id = i;  ──────────── idの前にオブジェクト名を付けない
strcpy(name, n);  ──── nameの前にオブジェクト名が付かない
```

次にsetdataの完成版を示す．

```
void Customer::setdata(int i, char *n)
{
      id = i;
      strcpy(name, n);
}
```

●show

このメンバー関数は，2つのメンバー変数の値を表示する．

```
void Customer::show()
{
      cout << "id = " << id << endl
      << "name = " << name << endl;
}
```

ここでもメンバー変数idとnameの前にオブジェクト名を付けていないことに注意.

ポイント

メンバー関数の中からメンバー変数を参照するときは
前にオブジェクト名を付けない

3.4◆顧客クラス（第3版）

メンバー関数の使い方を示したサンプルプログラムを次に示す.

【サンプルプログラム】

これは，第1節のサンプルプログラムにメンバー関数を追加したものである．このクラス宣言は，後の節で修正を受けることになる.

```
#include <iostream.h>

class Customer {
public:
    int id;
    char name[20];
    void setdata(int i, char *n); ――――メンバー関数を追加
    void show();
};

void Customer::setdata(int i, char *n)
{
    id = i;
    strcpy(name, n);
}

void Customer::show()
{
    cout << "id = " << id << endl
         << "name = " << name << endl;
}

int main()
{
    Customer c;

    c.setdata(100, "Stallman"); ――――メンバー関数を呼び出す
```

```
    c.show();

    return 0;
}
```

【実行例】

```
% a.out
100 ─────────────── 顧客番号
Stallman ────────── 顧客名
% _
```

4 アクセス制御

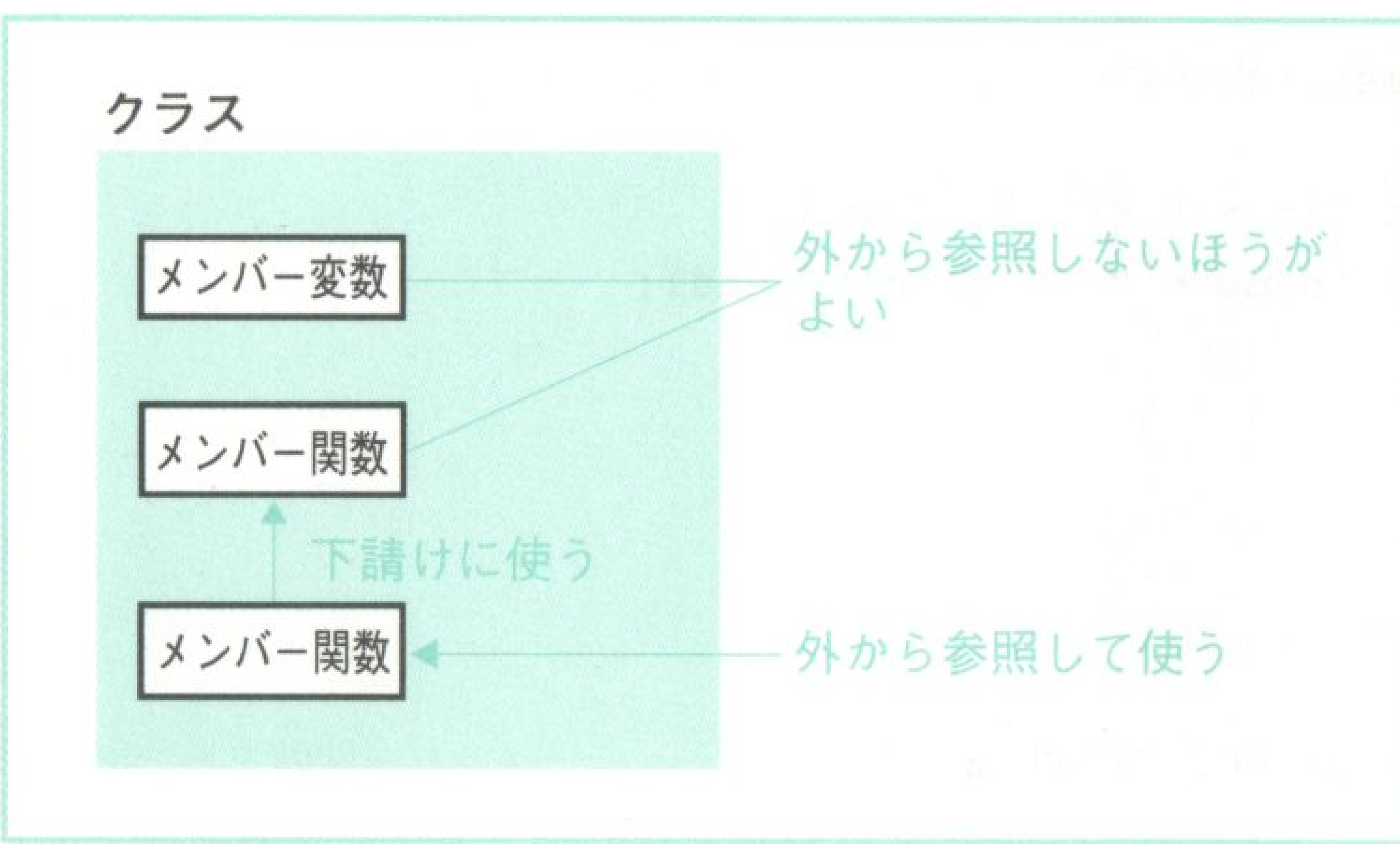
第1節のサンプルプログラムは，publicというキーワードを付けないとコンパイルできなかった．この節で，その理由を説明する．

4.1◆メンバーの可視性

メンバー変数は，できるだけクラスの外から参照しないほうがよい．一方，メンバー関数は，外から参照してもかまわない．ただし，メンバー関数が補助に使用する関数は，外から参照する必要はない（参照させない）．

このように，クラスのメンバーには，外から参照してよいものと，参照しないほうがよいものの2種類がある．

図20-2▶



4.2◆セクション

C++は，外から参照されたくないメンバーを保護するための仕掛けを用意している．すなわち，メンバーをいくつかのセクションに分け，それぞれのセクションについて，クラスの外から参照してよいかどうかを指定することができる．セクションには，次のようなものがある．

privateセクション

このセクションにあるメンバーをprivateメンバーという
privateメンバーはクラスの外から参照できない
privateメンバーはクラスのメンバー関数から参照可能

publicセクション

このセクションにあるメンバーをpublicメンバーという
publicメンバーはどこからでも参照可能

クラス内部は，デフォルトではprivateセクションになる．

```
class Customer {
    // 以下の部分はprivateセクション
    …………        // これらのメンバーはprivateメンバー
    …………
    …………
};
```

前章のサンプルプログラムでは，すべてのメンバーがデフォルトのprivateメンバーになっていた．このため，クラスの外からメンバーを参照するとエラーになる．これが，前章のサンプルプログラムがコンパイルエラーになった理由である．

4.3◆アクセス制御キーワード

private以外のセクションを作るには，**アクセス制御キーワード**を使う．たとえば，publicセクションを作るには，publicキーワードを使って次のようにする．

```
class Customer {
    …………  ←―――― privateセクション
```

```
    ............
public:
    ............  ◀──────── publicセクション
    ............
};
```

クラス内部は，デフォルトではprivateセクションになる．しかし，privateセクションであることを明示するためにprivateキーワードを使ってもよい．

```
class Customer {
private  ◀──────── privateセクションであることを明示
    ............
    ............
public:  ◀──────── 以下，publicセクション
    ............
    ............
};
```

次は，第3.4節のサンプルプログラムで示したCustomerのクラス宣言である．

```
class Customer {
public:
    int id; ──────── メンバー変数をpublicにしている
    char name[20];
    void setdata(int i, char *n);
    void show();
};
```

この宣言には欠点がある．なぜならメンバー変数をpublicセクションに置いているからである．メンバー変数は，クラスの外から参照させないほうがよいので，privateメンバーにしたほうがよい．そこで，次のようにprivateセクションに移動する．

```
class Customer {
    int id; ──────── メンバー変数をprivateにする
    char name[20];
public:
    void setdata(int i, char *n);
    void show();
};
```

あるいは，次のようにprivateキーワードを置いてprivateメンバーであることを明示してもよい．

```
class Customer {
private   ← privateセクションであることを明示
      int id;
      char name[20];
public:
      void setdata(int i, char *n);
      void show();
};
```

以下に，アクセス制御キーワードをまとめる．

キーワード	意味
private	privateセクションの始まりを示す
public	publicセクションの始まりを示す

注意　アクセス制御キーワードは，このほかにもいくつかある．

5 インラインメンバー関数

メンバー関数は，2つの方法で定義することができる．

① クラスの中で宣言と同時に定義する
② クラスの外で定義する

5.1◆クラスの外で定義する

第3節で説明した方法は，②によるものである．

```
class Customer {
private
```

```
        int id;
        char name[20];
    public:
        void setdata(int i, char *n);  ──── クラス内にはヘッダだけを置く
        void show();
    };

    // 実際の関数定義はクラスの外で行っている
    void Customer::setdata(int i, char *n)
    {
        ............
    }

    void Customer::show()
    {
        ............
    }
```

5.2◆インラインメンバー関数

このように記述する代わりに，クラスの中で宣言と同時に関数の本体を定義することもできる．次に例を示す．

```
    class Customer {
        ............
        void setdata(int i, char *n) { ◀──── 関数定義をクラス内で
            id = i;
            strcpy(name, n);
        }
        ............
    };
```

メンバー関数をクラス内で定義すると，そのメンバー関数は自動的に**インライン関数**になる．

インライン関数については，すでに第1部の第12章第6節で説明した．それと同じようにインラインのメンバー関数を呼び出すと，その位置に関数本体のコード

が直接埋め込まれる．したがって，関数呼び出し時におけるオーバーヘッドを最小にすることができる．このような書き方は，数行程度の小さい関数に向いている．

5.3◆顧客クラス（第4版）

次に示すのは，第3.4節のサンプルプログラムをインライン関数で書き換えたものである．また，メンバー変数をprivateセクションに移動している．

【サンプルプログラム】

```
#include <iostream.h>

class Customer {
    int id;  ―――― privateセクション
    char name[20];
public:
    void setdata(int i, char *n) {  ―― インライン関数
        id = i;
        strcpy(name, n);
    }

    void show() {
        cout << "id = " << id << endl
             << "name = " << name << endl;
    }
};

int main()
{
    Customer c;

    c.setdata(100, "Stallman");
    c.show();

    return 0;
}
```

【実行例】

```
% a.out
100  ―――― 顧客番号
Stallman  ―――― 顧客名
% _
```

6 何をクラスにするか？

以上，クラスの基本を説明してきた．この章の最後として，「何をクラスにすればよいか？」の指針を示す．

「オブジェクト」は，クラス型変数を宣言するなどして，クラス型のデータを実際にメモリ上に割り当てるものである．オブジェクトは変数と関数をメンバーに持つ．

クラスはユーザー定義型なので，どの変数と関数をメンバーにするかは，ユーザーの自由である．したがって，相互に何の関係もないメンバーどうしをいっしょのクラスにしても（文法上は）かまわない．

しかし，それでは役に立つクラスを作ることはできない．関連のある変数と関数を1つのクラスにまとめるのが普通である．つまり，関連のあるデータと関数を1つの「もの」として扱おうとするのが「オブジェクト」である．

関連する複数のデータと関数をまとめ
1つのオブジェクトとして扱う

――これが，オブジェクト指向プログラミングへの第一歩である．

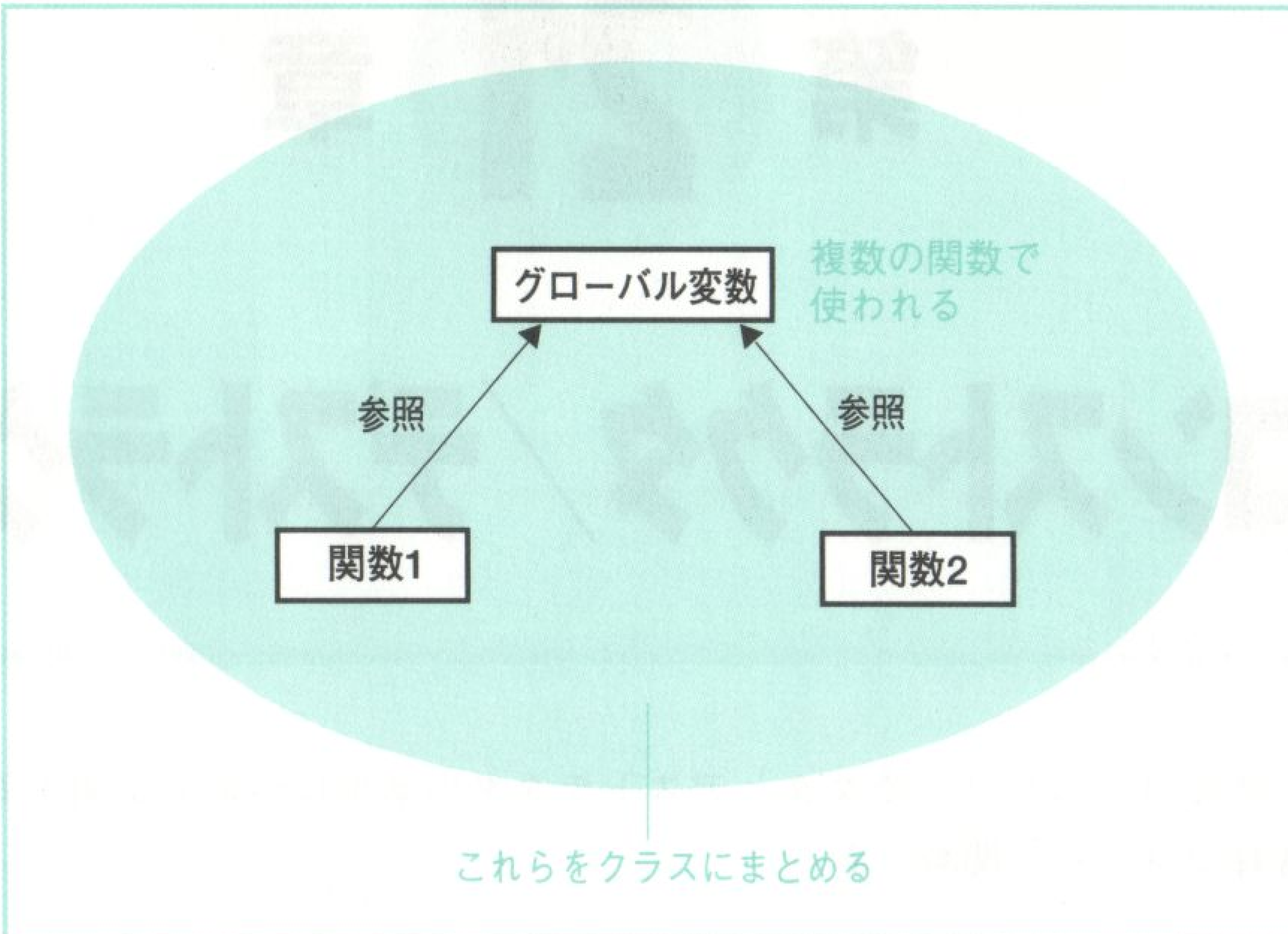
Cなどの非オブジェクト指向言語からオブジェクト指向言語に入ってくると，最初は何をクラスにしてよいか迷うかもしれない．そのための技巧もいろいろ発表されている．最初は，次のように始めるのがわかりやすい．

グローバル変数があって，
そのグローバル変数を複数の関数で参照しているなら
それらを1つのオブジェクトにする

グローバル変数を複数の関数が参照するということは，それらが関連する仲間であるからにほかならない．

ここから出発して経験を積んでいけば，やがて，何をクラスにすればよいかわかってくる．

図20-3▶



第21章 コンストラクタ／デストラクタ

この章では，コンストラクタとデストラクタの基本について説明する．この2つは，特殊なメンバー関数である．

コンストラクタ――――――オブジェクトを初期化する
デストラクタ――――――オブジェクトの後処理をする

ここでいう初期化と後処理は，C++コンパイラ内部から見た場合と，それを使うプログラマーから見た場合とでは意味が異なる．前者については「**動的オブジェクト**」の章（第25章）で説明する．
この章では，プログラマーの立場から見たオブジェクトの初期化と後処理を扱う．

1 オブジェクトの初期化

C++を設計したストラウストラップは初期化にこだわったようで，オブジェクトについても初期化の忘れを防止する仕掛けを用意した．それが，コンストラクタの自動実行である．
コンストラクタは，特殊なメンバー関数である．通常，その中にオブジェクトを初期化するためのコードを書く．したがって，コンストラクタを実行することによって，オブジェクトを初期化することができる．

ただし，コンストラクタがあったとしても，それでオブジェクトの初期化の忘れを防止できるわけではない．コンストラクタが実行されて初めてオブジェクトが初期化される．問題は，ここにある．コンストラクタが実行されるか，されないか？

C++は，（たとえユーザーがコンストラクタを呼び出すのを忘れても）自動的に実行される．C++に慣れた者から見ると当り前のことのように思うが，世の中には，コンストラクタが自動的に実行されない言語も存在するのである．

C++ではコンストラクタが自動実行される——ここに意義がある．以下，具体例を掲げて，この点を説明していく．

1.1◆クラスDiv

いま，次のようなクラスDivがあるとする．

```
class Div {
    ........................
};
```

Divは，2数の割り算を行うクラスである．割り算を行うのにわざわざクラスにする必要もないが，説明のために用意した．Divは，2つのメンバー変数を持っている．

```
class Div {
    int val1, val2; ──────── メンバー変数
    ........................
};
```

これらの2数が，被除数（割られる数）と除数（割る数）である．この2つのメンバー変数は，メンバー関数setにより初期化される．

```
class Div {
    int val1, val2;
public:
    void set(int n1, int n2); ──── 初期化用のメンバー関数
    ........................
};
```

メンバー関数setの定義は次のようになっている．

```
void Div::set(int n1, int n2)
{
```

```
        val1 = n1;
        val2 = n2;
    }
```

次のメンバー関数getを呼び出すことで商を得ることができる．

```
    class Div {
        int val1, val2;
    public:
        void set(int n1, int n2);
        int get(); ───────── 結果を得るためのメンバー関数
    };
```

メンバー関数getの定義は，次のようになっている．

```
    int Div::get()
    {
        return val1 / val2 ;
    }
```

とりあえず，ここまでをテストするため，次のサンプルプログラムを示す．

【サンプルプログラム】

これは，クラスDivの使い方を示すものである．単純な除算を行っている．

```
#include <iostream.h>

class Div {
    int val1, val2;
public:
    void set(int n1, int n2);
    int get();
};

void Div::set(int n1, int n2)
{
    val1 = n1;
    val2 = n2;
}

int Div::get()
{
```

```
    return val1 / val2 ;
}

int main()
{
    Div d;

    d.set(1200, 4);
    cout << d.get() << endl;

    return 0;
}
```

【実行例】

```
% a.out
300 ――――――――――――――――――――――――――― 1200 / 4が計算される
% _
```

1.2◆メンバー変数の初期化を忘れた例

このサンプルプログラムで重要な部分は，関数mainの中の次のところである．

```
d.set(1200, 4); ―――――――――― メンバー変数を初期化してから
cout << d.get() << endl; ――――― 除算を行っている
```

ここで，d.set(1200, 4); を忘れたらどうなるだろう？ この結果は悲惨である．次のようにcoreを吐いて死んでしまう（d.setの行をコメントにしてコンパイル／実行する）．

図21-1▼

```
% a.out
Floating point exception (core dumped) ―――――― コアダンプされる
% _
```

これは，ひとえにメンバー変数の初期化を忘れて，その他の処理を強行したためである．このように，オブジェクトの初期化は重要である．

ちなみに，この例では，ただ除算を行っているだけなので，メンバー関数getを書き換え，次のようにval2が0でないかをチェックするだけでcoreを吐くのを防ぐことができる．

```
int Div::get()
{
    if (val2 != 0) ――――――― 除数が0でないかを確かめてから
        return val1 / val2 ; ――― 除算を行う
    else
        return 0;
}
```

あるいは，次のように三項演算子を使ってもよい．

```
int Div::get()
{
    return (val2) ? val1 / val2 : 0;
}
```

このように改良しても，オブジェクトの初期化を忘れれば（エラーは発生しないものの）予期せぬ結果が得られることに変わりはない．

2 コンストラクタ

前節で登場したクラスDivは，メンバー変数を初期化するのにsetというメンバー関数を使用していた．通常，オブジェクトを初期化するためのコードは，**コンストラクタ**の中に書く．この節では，クラスDivにコンストラクタを導入し，コンストラクタの使い方をまとめる．

2.1◆コンストラクタの宣言

コンストラクタ（*constructor*）は，メンバー関数の一種である．しかし，次の点で，普通のメンバー関数とは異なる．

コンストラクタ

① 関数の名前をクラス名と同じにする
② 必ずpublicセクションで宣言する
③ 型を指定することはできない
　したがって値を返すことはできない
④ 引数をとるのは自由である

この規則に従って，クラスDivにコンストラクタを導入する．まず規則①によって，コンストラクタの名前はDivになる．

```
class Div {
    …………
    Div();  ────── コンストラクタ
    …………
};
```

規則②に従って，コンストラクタをpublicセクションに置く．これは，コンストラクタがクラスの外から呼び出されるからである．コンストラクタをprivateセクションに置くと，コンパイルエラーになる．

```
class Div {
    …………
public:  ────── コンストラクタは
    Div();         publicセクションに置く
    …………
};
```

規則③に従って，コンストラクタには型を指定しない．上記では型を指定していないので，これでコンストラクタの宣言は完成である．引数をとることもできるが，Divは引数なしのコンストラクタにした．

2.2◆コンストラクタの定義

次にコンストラクタの本体を定義する．書き方は，通常のメンバー関数の場合と同じで，コンストラクタの前に「クラス名::」を付ける．

これを付けるのを忘れないこと!!

```
Div::Div()
{
    ............
}
```

コンストラクタの中では，メンバー変数を初期化するためのコードを書く．直接メンバー変数に値を代入してもよいが，すでにメンバー変数を初期化するためのメンバー関数setを持っているので，setを呼び出してもよい．

```
Div::Div()
{
    set(1200, 4);   ――― setを使ってメンバー変数を初期化する
}
```

なお，コンストラクタが実行されたことがわかるように，ここでは次のコードも書いておく．

```
Div::Div()
{
    cout << "Div::Div() running.." << endl;   ◀― この文を追加
    set(1200, 4);
}
```

これでコンストラクタDivの定義は終わりである．

2.3◆コンストラクタの実行

クラスにコンストラクタを追加しても，オブジェクトの使い方は変わらない．普通にクラス型変数を宣言すればよい．

```
Div d;   ――― (1)
```

コンストラクタを呼び出す必要もない．コンストラクタは，オブジェクトが作られるときに自動的に実行される．それを次のサンプルプログラムで確かめる．

【サンプルプログラム】

```
#include <iostream.h>

class Div {
    int val1, val2;
public:
    Div();  ──────────────── コンストラクタ
    void set(int n1, int n2);
    int get();
};

Div::Div() ──────────────── コンストラクタ本体の定義
{
    cout << "Div::Div() running..." << endl;
    set(1200, 4);
}

void Div::set(int n1, int n2)
{
    val1 = n1;
    val2 = n2;
}

int Div::get()
{
    return (val2) ? val1 / val2 : 0;
}

int main()
{
    Div d;
    cout << d.get() << endl;
    return 0;
}
```

【実行例】

```
% a.out
Div::Div() running... ◀──────── コンストラクタが呼び出されている
300
% _
```

ここで注目したいのが関数mainである．

```
int main()
{
    Div d;
    cout << d.get() << endl;
    …………
}
```

Div d;でオブジェクトを作成した後，メンバー変数を初期化することなく，ただちにd.get()を呼び出している．にもかかわらず，正しくオブジェクトの値が表示されている（第1.2節「**メンバー変数の初期化を忘れた例**」のサンプルプログラムと比較せよ）．

これは，オブジェクトが作成されるとき自動的にコンストラクタが実行され，メンバー変数を初期化しているからである．

2.4◆引数付きのコンストラクタ

第2.3節のコンストラクタは，メンバー変数を1200と4で初期化した．

```
Div::Div()
{
    cout << "Div::Div() running..." << endl;
    set(1200, 4);  ←── ここ
}
```

これでは，いつも1200÷4しか計算できない．

コンストラクタは，型を指定することはできないが，引数を指定することはできる．コンストラクタDivに2つの引数を追加し，任意の値でメンバー変数を初期化できるようにする．新しいDivは次のようになる．

```
class Div {
    …………
public:
    Div(int n1, int n2);  ←── 引数を指定する
    …………
};
```

```
Div::Div(int n1, int n2)
{
    cout << "Div::Div(int n1, int n2) running.." << endl;
    set(n1, n2);  ← 引数でメンバー変数の初期化
}
```

引数付きのコンストラクタがある場合は，クラス型変数を宣言するとき，次のようにして引数をコンストラクタに渡す．

```
Div d(2000, 25);
```

これらの引数がコンストラクタに渡される

以上をテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

class Div {
    int val1, val2;
public:
    Div(int n1, int n2);
    void set(int n1, int n2);
    int get();
};

Div::Div(int n1, int n2)
{
    cout << "Div::Div(int n1, int n2) running..." << endl;
    set(n1, n2);
}

void Div::set(int n1, int n2)
{
    val1 = n1;
    val2 = n2;
}

int Div::get()
{
```

```
    return (val2) ? val1 / val2 : 0;
}

int main()
{
    Div d(2000, 25);
    cout << d.get() << endl;

    return 0;
}
```

【実行例】

```
% a.out
Div::Div() running...
80
% _
```

3 コンストラクタの多重定義

1つのクラスに**複数のコンストラクタ**を定義することができる．これを**コンストラクタの多重定義**という．コンストラクタを多重定義すると，いろいろな方法でオブジェクトを初期化することができる．この節では，コンストラクタの多重定義について説明する．

3.1◆2つのコンストラクタを持つクラス

たとえば次は，メンバー変数を1つ持つクラスFooである．

```
class Foo {
    int val; ———————— メンバー変数
    ............
};
```

このクラスを2つの方法で初期化できるようにする．1つは直接int型のデータを与える方法であり，もう1つは数を表す文字列を与える方法である．すなわち，このクラスは次の2つのコンストラクタを持つ．

```
class Foo {
    int val;
public:
    Foo(int n);  ―――――― 第1コンストラクタ
    Foo(char *s); ――――― 第2コンストラクタ
};

Foo::Foo(int n) ―――――― 引数は整数
{
    val = n;
}

Foo::Foo(char *s) ――――― 引数は文字列
{
    val = atoi(s);――――― 文字列を整数に変換
}
```

3.2◆コンストラクタの選択

複数のコンストラクタがあるとき，どちらのコンストラクタが実行されるかは，オブジェクトを作成するときに与える引数で決まる．たとえば，次のようにf1，f2の2つのオブジェクトを作ったとする．

```
Foo f1(1000), f2("-8100");
```

f1の方は整数で引数を与えているのでFoo（int n）の方のコンストラクタが実行される．また，f2の方は文字列で引数を与えているのでFoo（char *s）の方のコンストラクタが実行される．

以上をテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>
#include <stdlib.h>

class Foo {
```

```
    int val;
public:
    Foo(int n);
    Foo(char *s);
    void show();
};

Foo::Foo(int n) ―――――――――――――― 第1コンストラクタ
{
    val = n;
}

Foo::Foo(char *s) ―――――――――――― 第2コンストラクタ
{
    val = atoi(s);
}

void Foo::show()
{
    cout << val << endl;
}

int main()
{
    Foo f1(1000), ――――――――――――― f1は第1コンストラクタが実行される
        f2("-8100"); ――――――――――― f2は第2コンストラクタが実行される

    f1.show();
    f2.show();

    return 0;
}
```

【実行例】

```
% a.out
1000 ―――――――――――――――――――― オブジェクトf1の値
-8100 ――――――――――――――――――― オブジェクトf2の値
% _
```

このように，1つのクラスに複数のコンストラクタを多重定義することができる．それらのコンストラクタは引数によって区別されるので，同じ引数を持つコンストラクタを多重定義することはできない．たとえ引数の名前を変えても型が同じ

なら同じ引数と見なされるので，引数の名前を変えただけではダメである．

4 デフォルトコンストラクタ

1つのクラスに定義できるコンストラクタの数は自由である．複数のコンストラクタを持つクラスもあれば，コンストラクタを1つも持たないクラスもある．

クラスにコンストラクタが1つも定義されていないときでも，裏で自動的にコンストラクタが作られる．これを**デフォルトコンストラクタ**（*default constructor*）という．

デフォルトコンストラクタは，引数がなく，関数本体も空のコンストラクタである．たとえばClassnameというクラスにコンストラクタがない場合，次のコンストラクタが自動的に作成される．

```
Classname::Classname() ―――― 引数はない
{ ―――――――――――――――― 関数の本体も空
}
```

オブジェクトが作られるとき，この空のコンストラクタが実行される．

本体が空の関数を実行しても何の意味もないように見える．しかし，これが意味があるのである．このことは，この章の最初に触れたことに関係する．すなわち，C++がそれを必要とするのである．

オブジェクトが作成されるとき，プログラマーからは見えない裏の初期化が必要である．その初期化コードは，コンストラクタが実行されるとき実行される．ゆえに，どのようなクラスにおいてもコンストラクタが実行される必要があるのである．

デフォルトコンストラクタは，コンストラクタを持たないクラスでも必ずコンストラクタを実行させるために作られる．

このように，C++のコードは，プログラムからは見えない隠されたコードが実行されることがある．この不透明さを嫌うプログラマーもいる．

5 デストラクタ

コンストラクタと対をなすのがデストラクタである．**デストラクタ**（*destructor*）は，オブジェクトの後処理を行う特殊なメンバー関数である．

5.1◆デストラクタの宣言

デストラクタは，メンバー関数の一種である．しかし，次の点で，普通のメンバー関数とは異なる．

デストラクタ

① クラス名と同じ名前で先頭に“~”（チルダ）を付ける
② 必ずpublicセクションで宣言する
③ 型を指定することはできない
　したがって値を返すことはできない
④ 引数を指定することもできない
⑤ 多重定義することもできない

デストラクタの宣言は，コンストラクタに似ている．外部から呼び出されるのでpublicセクションで宣言する点も，型を指定することができない点も似ている．しかし，名前の前に~（チルダ）を付けるところが異なる．

```
class Dest {
public:  ―――――――――― publicセクションで宣言する
    Dest(); ――――――――― コンストラクタ
    ~Dest(); ―――――――― デストラクタ
};
```

コンストラクタは引数を指定することができたが，デストラクタはできない．したがって，引数による区別ができないので，多重定義することはできない．

次にデストラクタの使い方を示すサンプルプログラムを示す．

【サンプルプログラム】

```
#include <iostream.h>

class Dest {
public:
    Dest();
    ~Dest();
};

Dest::Dest()
{
  cout << "constructor  running ..." << endl;
}

Dest::~Dest()
{
  cout << "destructor running ..." << endl;
}

int main()
{
    Dest d;
    cout << "main() running ..." << endl;

    return 0;
}
```

【実行例】

```
% a.out
constructor  running ...
main() running ...
destructor running ...  ←── デストラクタが実行されている
% _
```

5.2◆デストラクタが実行される時期

このサンプルプログラムは，コンストラクタとデストラクタだけを持つクラスDestを使用している．メンバー変数すら持っていない．

```
class Dest {
public:
      Dest(); ――――――――― コンストラクタ
      ~Dest(); ――――――――― デストラクタ
};
```

関数mainでは，Destクラスのオブジェクトを作成し，coutでmainが実行されていることを示しているだけである．

```
int main()
{
      Dest d; ――――――――― オブジェクトを作成
      cout << "main() running ..." << endl;

      return 0;
}
```

デストラクタを呼び出すことはしていない．しかし，実行結果を見ると，最後にデストラクタが実行されている．

```
% a.out
constructor  running ...
main() running ...
destructor running ... ◀――――― デストラクタが実行されている
% _
```

デストラクタは，次の時点で自動的に実行される．

① 変数の有効範囲が終了するとき
② オブジェクトのメモリが解放されるとき

この例では，オブジェクトdは関数mainの中で宣言されている．したがって，関数mainが終了すると同時にdの有効範囲が終了するので，デストラクタが実行される．

このようにデストラクタは，適切な時期に自動的に実行される．初期化と後処理が対になっているとき（たとえばメモリの確保と解放），後処理をデストラクタの中に書いておくと，確実に後処理を行うことができる．

第22章 継承

継承は，オブジェクトに階層構造を作る．継承により，祖先に当たるオブジェクトを起点に，その一族を統一した形で扱うことができる．

最初にオブジェクトを継承する方法を説明する．続いて**メンバーの可視性**を，セクションと継承修飾子との関係から説明する．最後に**コンストラクタ**が継承にどう作用するかを説明する．

1 継承

この節では，既存のクラスを継承して新しいクラスを作成する方法を説明する．

1.1◆果物クラス

話の糸口として，**果物クラス**から見ていく．この章は，この果物クラスを中心に話を進める．

```
class Fruit1 {
    ............
};
```

果物クラスは，フルーツの数を管理する．ただし，説明を簡単にするため，Fruit1はappleというメンバー変数だけを持っている．これは，リンゴの数を表現する．個数なので，appleは負にはならない．

```
class Fruit1 {
    int apple;  ―――――― リンゴの個数  apple >= 0
    ............
};
```

説明の都合上，Fruit1はコンストラクタを持たない（継承とコンストラクタとの関係は，この章の最後のところで説明する）．代わりにsetとshowの2つのメンバー関数を持つ．

```
class Fruit1 {
    int apple;
public:
    void set(int n);
    void show();
};
```

メンバー関数のsetは，リンゴの個数をメンバー変数のappleに設定する．そのとき，もし負数が指定されたら，個数を0に置き換える．

```
void Fruit1::set(int n)
{
    apple = (n < 0) ? 0 : n;
}
```

三項演算子を使っているのでわかりにくいかもしれない．次のように書き換えてもよい．

```
void Fruit1::set(int n)
{
    if (n < 0)
        apple = 0;  ―――――― 負数を指定された場合
    else
        apple = n;
}
```

メンバー関数のshowは，リンゴの個数を表示する．

```
void Fruit1::show()
{
```

```
        cout << "apple = " << apple << endl;
    }
```

次にクラスFruit1の使用例を示す.

【サンプルプログラム】

```
#include <iostream.h>

class Fruit1 {
    int apple;  ―――――――――――― リンゴの数
public:
    void set(int n);
    void show();
};

void Fruit1::set(int n)  ―――――――― リンゴの数を設定する
{
    apple = (n < 0) ? 0 : n;
}

void Fruit1::show()  ―――――――――― リンゴの数を表示する
{
    cout << "apple = " << apple << endl;
}

int main()
{
    Fruit1 f;

    f.set(1000);  ――――――――――― リンゴの数を1000個に設定
    f.show();

    return 0;
}
```

【実行例】

```
% a.out
apple = 1000  ―――――――――――― 現在リンゴの数は1000個
% _
```

1.2◆基本クラスと派生クラス

前項でクラスFruit1を示した.

```
class Fruit1 {
    int apple;
public:
    void set(int n);
    void show();
};
```

これによく似たFruit2というクラスが必要になったとする.

```
class Fruit2 {
    int apple;
    int orange;
public:
    void set(int n1, int n2);
    void show();
};
```

Fruit2はFruit1にorange（オレンジの個数）というメンバー変数を追加したものである．メンバー関数もほとんど同じものを持っている．setはメンバー変数が増えたので，引数が増えている．showは見かけは同じだが，新しいshowはオレンジの個数も表示するように拡張される.

このようにFruit1とFruit2はよく似ている．ほとんど同じなのに，もう一度同じような定義を書くのは無駄に見える.

あるクラスにメンバーを追加・修正して新しいクラスを作るとき，**継承**という機能を使うことができる．継承を使えば，同じメンバーを繰り返し宣言する無駄を省くことができる.

新しいクラス元になるクラスを**基本クラス**という．また，基本クラスから作られる新しいクラスを**派生クラス**という.

基本クラス（*base class*）

↑ 継承

派生クラス（*derived class*）

注意　前ページ下の図では矢印の向きが逆に見えるかもしれない．オブジェクト指向では，派生クラスから基本クラスに向かって矢印を引き，継承関係を示す．

派生クラスは，基本クラスのメンバーを自動的に持つことができる．

1.3◆派生クラスの宣言

派生クラスを宣言するには，クラス名の後にコロンを付け，その後ろに基本クラス名を付ける．

```
class 派生クラス名 : 基本クラス名 {
    ......................
};
```

ここでは，Fruit2をFruit1から作るので，次のように宣言する．

```
class Fruit2 : Fruit1 {
    ......................
};
```

この段階でFruit2はFruit1のすべてのメンバーを自動的に持つことになる．したがって，宣言することなく，appleやset()やshow()を持っている．

{} の中には，追加するメンバーだけを並べる．

```
class Fruit2 : Fruit1 {
    int orange; ―――――追加するメンバー
};
```

これで，Fruit2にorangeというメンバー変数が追加される．ここには書いていないが，もちろんappleというメンバー変数も持っていることになる．

Fruit2は，そのほかにも，setとshowのメンバー関数を持っている．しかし，これらの関数は，仕様が変更になるので宣言に加える．

```
class Fruit2 : Fruit1 {
    int orange;
public:
    void set2(int n1, int n2);
```

```
        void show2();
};
```

ここで基本クラスのsetやshowと名前が衝突しないようにメンバー関数の名前を

```
set  →set2
show→show2
```

のように変更した．しかし，関数の名前を変えずに上書きすることもできる．それについては，第5節「メンバー関数のオーバーロード」のところで改めて説明する．

新しいset2とshow2の定義は次のようになる．

```
void Fruit2::set2(int n1, int n2)
{
    apple = (n1 < 0) ? 0 : n1;
    orange = (n2 < 0) ? 0 : n2;
}

void Fruit2::show2()
{
    cout << "apple = " << apple << endl;
    cout << "orange = " << orange << endl;
}
```

以上をまとめると，次のサンプルプログラムが出来上がる．

【サンプルプログラム】

```
#include <iostream.h>

class Fruit1 {  ―――― 基本クラスの宣言
    int apple;
public:
    void set(int n);
    void show();
};

void Fruit1::set(int n)
{
    apple = (n < 0) ? 0 : n;
```

```
}

void Fruit1::show()
{
    cout << "apple = " << apple << endl;
}

class Fruit2 : Fruit1 { ──────────── 派生クラスの宣言
    int orange;  // orange > 0
public:
    void set2(int n1, int n2);
    void show2();
};

void Fruit2::set2(int n1, int n2)
{
    apple = (n1 < 0) ? 0 : n1;
    orange = (n2 < 0) ? 0 : n2;
}

void Fruit2::show2()
{
    cout << "apple = " << apple << endl;
    cout << "orange = " << orange << endl;
}

int main()
{
    Fruit2 f;

    f.set2(1000, 2300);
    f.show2();

    return 0;
}
```

1.4◆コンパイルエラー

残念ながら，前項のサンプルプログラムは，コンパイルエラーになる．

図22-1▼

```
% g++ go.C
go.C: In method `void Fruit2::set2(int, int)':
go.C:30: member `apple' is private
go.C: In method `void Fruit2::show2()':
go.C:36: member `apple' is private
% _
```

これは，Fruit2をFruit1からprivateに継承したからである．その意味を次の節で説明する．

2 privateな継承

派生クラスを宣言すると，デフォルトでは**privateな継承**になる．privateな継承は，次の点で参照が厳しく制限される．

- 派生クラスのメンバー関数から基本クラスのprivateメンバーを参照できない
- 基本クラスのpublicメンバーが派生クラスではprivateメンバーになってしまう

この節では，以上の点について説明する．

2.1◆継承修飾子

派生クラスを宣言するとき，基本クラス名の前に**継承修飾子**を指定することができる．

ここ
↓
```
class 派生クラス名 : 継承修飾子 基本クラス名 {
    ......................
};
```

継承修飾子には，privateとpublicがある．

> **注意** privateとpublicというキーワードは，メンバーのアクセス制御にも使用した．継承修飾子は，それとは異なるので混同しないように．

2.2◆privateな継承

継承修飾子を省略すると，privateが指定されたものと見なされる（privateな継承）．たとえば，次のFruit2はFruit1をprivateに継承している．

```
class Fruit2 : Fruit1 {
    ............
};
```

この宣言は，次と同じである．

```
class Fruit2 : private Fruit1 {
    ............
};
```

privateな継承では，基本クラスの参照が厳しくなる．たとえば，基本クラスのprivateメンバーの扱いがそうである．それを次に説明する．

privateセクションのメンバーは，クラスの外からは参照できない．しかし，クラスの中（要するにメンバー関数の中）からは参照できる．たとえば，次のappleはprivateメンバーなのでクラスの外からは参照できないが，内部のsetやshowからは参照できる．

```
class Fruit1 {
    int apple;
public:
    void set(int n);
    void show();
};
```

ここまでは，当然である．ところが，これをprivateに継承すると，たとえメンバー関数であってもappleを参照できなくなる．

```
void Fruit2::set2(int n1, int n2)
{
      apple = (n1 < 0) ? 0 : n1;        appleは基本クラスの
      orange = (n2 < 0) ? 0 : n2;       privateメンバーなので
}                                       参照できない

void Fruit2::show2()
{
      cout << "apple = " << apple << endl;      これもダメ
      cout << "orange = " << orange << endl;
}
```

privateな継承では，基本クラスのprivateメンバーを参照できないのである．第1.3節のサンプルプログラムがコンパイルエラーになったのは，このためである．

2.3◆基本クラスのprivateメンバー

privateに継承した派生クラスで，基本クラスのprivateメンバーを参照するには，クラスの外と同じように，基本クラスのメンバー関数を使えばよい．publicなメンバー関数なら，派生クラスでも使うことができる．

そこで，set2とshow2を次のように書き換える．

```
void Fruit2::set2(int n1, int n2)
{
      set(n1);      ここ
      orange = (n2 < 0) ? 0 : n2;
}

void Fruit2::show2()
{
      show();      ここ
      cout << "orange = " << orange << endl;
}
```

setとshowならpublicメンバーなので，派生クラスからも参照することができる．これで，第1.3節のサンプルプログラムをコンパイルできるようになる．

【サンプルプログラム】

これは，第1.3節のサンプルプログラムのset2とshow2を書き換えてコンパイルできるようにしたものである．

```
#include <iostream.h>

class Fruit1 {
    int apple;    // apple >= 0
public:
    void set(int n);
    void show();
};

void Fruit1::set(int n)
{
    apple = (n < 0) ? 0 : n;
}

void Fruit1::show()
{
    cout << "apple = " << apple << endl;
}

class Fruit2 : Fruit1 {
    int orange;  // orange > 0
public:
    void set2(int n1, int n2);
    void show2();
};

void Fruit2::set2(int n1, int n2)
{
    set(n1);
    orange = (n2 < 0) ? 0 : n2;
}

void Fruit2::show2()
{
    show();
    cout << "orange = " << orange << endl;
}

int main()
```

```
{
    Fruit2 f;

    f.set2(1000, 2300);
    f.show2();

    return 0;
}
```

【実行例】

```
% a.out
apple = 1000
orange = 2300
% _
```

2.4◆publicメンバーがprivateに

privateな継承で厳しいのは，基本クラスのprivateメンバーを参照できないだけではない．なんと，publicメンバーでさえ参照できなくなってしまうのである．privateな継承では，基本クラスのpublicメンバーがprivateメンバーになってしまう．つまり，クラスの外から基本クラスのpublicメンバー関数が使えなくなる．

それを次の例で確かめる．

```
int main()
{
    Fruit2 f;  ―― Fruit2はFruit1のprivateな派生クラス

    f.set2(1000, 2300);
    f.show2();

    return 0;
}
```

これは，Fruit2型のオブジェクトを作成し，値を設定して表示している．ここまでは問題ない．前の項のサンプルプログラムで確かめたとおりである．

次にappleの個数を変更したいとする．appleの個数だけを変更するなら，基本クラスFruit1のsetが使える．setはpublicメンバーだから使えるはずである．

```
int main()
{
    Fruit2 f;

    f.set2(1000, 2300);
    f.show2();
    f.set(1500);        appleの個数を変更
    f.show2();

    return 0;
}
```

ところが，プログラムをこのように書き換えると，コンパイルエラーになる．

図22-2▼

```
% g++ go.C
go.C: In function `int main()':
go.C:11: method `void Fruit1::set(int)' is from private base class
go.C:45: within this context
% _
```

privateな継承では，クラスの外から基本クラスのpublicメンバー関数が使えなくなってしまうのである．この状況を改善するには，次の節で説明するpublicな継承を使う．

3 publicな継承

派生クラスを宣言するとき，基本クラス名の前にpublicを付けると**publicな継承**になる．

```
                    ここ
class 派生クラス名 : public 基本クラス名 {
    ......................
};
```

基本クラスを継承して派生クラスを作るとき，通常はpublicな継承を使う．前節で紹介したprivateな継承を使うことは稀である．

3.1◆publicな継承

継承修飾子のpublicを付けると，アクセス制御が変化しない．すなわち，publicメンバーは，派生クラスでもpublicメンバーになる．しかし，privateな継承では，そうはならなかった（第2.4節「publicメンバーがprivateに」を参照）．

第2.4節「publicメンバーがprivateに」のサンプルプログラムをpublicな継承に変えて，コンパイルできるようにする．

```
class Fruit2 : public Fruit1 {
        int orange;
public:
        void set2(int n1, int n2);
        void show2();
};
```

ここ（← public）

Fruit2をこのように修正すれば，基本クラスのpublicメンバーを参照できるようになる．

```
int main()
{
        Fruit2 f;

        f.set2(1000, 2300);
        f.show2();
        f.set(1500);
        f.show2();

        return 0;
}
```

setは基本クラスのpublicメンバー
今度はコンパイルエラーにならない

【サンプルプログラム】

これは，第2.4節「publicメンバーがprivateに」のサンプルプログラムをpublicな継承に変えてコンパイルできるように修正したものである．

```
#include <iostream.h>

class Fruit1 {
    int apple;
public:
    void set(int n);
    void show();
};

void Fruit1::set(int n)
{
    apple = (n < 0) ? 0 : n;
}

void Fruit1::show()
{
    cout << "apple = " << apple << endl;
}

class Fruit2 : public Fruit1 {
    int orange;
public:
    void set2(int n1, int n2);
    void show2();
};

void Fruit2::set2(int n1, int n2)
{
    set(n1);
    orange = (n2 < 0) ? 0 : n2;
}

void Fruit2::show2()
{
    show();
    cout << "orange = " << orange << endl;
}

int main()
{
```

```
    Fruit2 f;

    f.set2(1000, 2300);
    f.show2();
    f.set(1500); ──────────リンゴの数を変える
    f.show2();

    return 0;
}
```

今度はエラーにならずにコンパイルすることができる．次に実行例を示す．

【実行例】

```
% a.out
apple = 1000
orange = 2300
apple = 1500 ──────────setを呼んでリンゴの数を変えている
orange = 2300
% _
```

3.2◆厳しい継承

publicな継承に変えても，やはり基本クラスのprivateメンバーを参照することはできない．

```
void Fruit2::set2(int n1, int n2)
{
    apple = (n1 < 0) ? 0 : n1; ──────appleは基本クラスの
    orange = (n2 < 0) ? 0 : n2;      privateメンバーなので
}                                    参照できない

void Fruit2::show2()
{
    cout << "apple = " << apple << endl; ────これもダメ
    cout << "orange = " << orange << endl;
}
```

実際，第3.1節のサンプルプログラムのset2とshow2を上のように書き換えると，次のようなコンパイルエラーになる．

図22-3▼

```
go.C: In method `void Fruit2::set2(int, int)':
go.C:30: member `apple' is private
go.C: In method `void Fruit2::show2()':
go.C:37: member `apple' is private
```

3.3◆protectedセクション

基本クラスのprivateメンバーを派生クラスから参照することができないと使いにくい．**protectedアクセス制御キーワード**を使うと，これが可能になる．

これまでアクセス制御キーワードには，privateとpublicがあった（第19章第6節を参照）．

private

class宣言の中でprivateセクションの開始を示す

privateメンバーは，クラスの外からは参照できない

public

class宣言の中でpublicセクションの開始を示す

publicメンバーは，クラスの外からも参照できる

ここに第3のキーワードとして，protectedを追加する．これにより，class宣言は3つのセクションに分かれることになる．

```
class Class_name {
private:  ―――― privateキーワードは省略可能
    // privateセクション
protected:
    // protectedセクション
public:
    // publicセクション
}
```

基本クラスにおいては，protectedメンバーはprivateメンバーと同じである．クラスのメンバー関数からは参照できるが，クラスの外からは参照できない．

protectedメンバーとprivateメンバーの違いが出るのは，publicな継承により作られた派生クラスにおいてである．すなわち，派生クラスから基本クラスのprivateメンバーを参照することはできないが，protectedメンバーなら参照できる．

protected

class宣言の中でprotectedセクションの開始を示す
protectedメンバーは，クラスの外からは参照できない
クラスのメンバー関数からは参照できる
publicな派生クラスのメンバー関数からも参照できる

このため，そのクラスを派生する予定があり，かつそのメンバーを派生クラスから参照させる予定があり，かつそのメンバーを外部から保護したいときは，そのメンバーを（privateメンバーではなく）protectedメンバーにするとよい．

たとえばFruit1のメンバー変数appleは，派生クラスからも参照させたい．そこで，protectedメンバーにする．

```
class Fruit1 {
protected:  ―――― キーワードを追加する
    int apple; ―――― protectedメンバーになる
public:
    void set(int n);
    void show();
};
```

こうすれば，派生クラスのFruit2からもappleを参照できるようになる．

```
void Fruit2::set2(int n1, int n2)
{
    apple = (n1 < 0) ? 0 : n1; ―――― 今度はエラーにならない
    orange = (n2 < 0) ? 0 : n2;
}

void Fruit2::show2()
{
```

```
        cout << "apple = " << apple << endl;———これもOK
        cout << "orange = " << orange << endl;
    }
```

【サンプルプログラム】

次は，前項のサンプルプログラムを上記のように書き換えたものである．

```
#include <iostream.h>

class Fruit1 {
protected:
    int apple;          // apple >= 0
public:
    void set(int n);
    void show();
};

void Fruit1::set(int n)
{
    apple = (n < 0) ? 0 : n;
}

void Fruit1::show()
{
    cout << "apple = " << apple << endl;
}

class Fruit2 : public Fruit1 {
    int orange;          // orange > 0
public:
    void set2(int n1, int n2);
    void show2();
};

void Fruit2::set2(int n1, int n2)
{
    apple = (n1 < 0) ? 0 : n1;
    orange = (n2 < 0) ? 0 : n2;
}

void Fruit2::show2()
{
    cout << "apple = " << apple << endl;
```

```
    cout << "orange = " << orange << endl;
}

int main()
{
    Fruit2 f;

    f.set2(1000, 2300);
    f.show2();
    f.set(1500);
    f.show2();

    return 0;
}
```

今度はエラーにならずにコンパイルすることができる．次に実行例を示す．

【実行例】

```
% a.out
apple = 1000
orange = 2300
apple = 1500 ―――――基本クラスのappleを参照してリンゴの数を変えている
orange = 2300
% _
```

4 アクセス制御と継承修飾子のまとめ

前の2つの節で説明した継承修飾子とアクセス制御についてまとめる．以下は，アクセス制御キーワードのまとめである．

キーワード	意味
private	privateセクションの始まりを示す このメンバーはクラス外からは参照できない 派生クラスのメンバー関数からも参照できない このクラスのメンバー関数からは参照できる

キーワード	意味
protected	protectedセクションの始まりを示す このメンバーはクラス外からは参照できない このクラスのメンバー関数からは参照できる private派生クラスのメンバー関数からは参照できない public派生クラスのメンバー関数からは参照できる
public	publicセクションの始まりを示す このメンバーはクラスの外からも参照できる

次に，アクセス制御キーワードと継承修飾子の関係をまとめる．

●デフォルトの派生＝privateな派生の場合

	基本クラスのメンバー		
	private	protected	public
クラス外から	×	×	○
基本クラスから	○	○	○
派生クラスから	×	×	○

○は参照可能，×は参照不可を表す

●publicな派生の場合

	基本クラスのメンバー		
	private	protected	public
クラス外から	×	×	○
基本クラスから	○	○	○
派生クラスから	×	○	○

○は参照可能，×は参照不可を表す

5 メンバー関数のオーバーロード

継承を説明するのに果物クラスFruit1，Fruit2を例に使ってきた．この2つのクラスは，setとshowというメンバー関数を持っている．ただし，名前の衝突を避けるため，2つのクラスで名前を変えてきた．

基本クラス（Fruit1）	派生クラス（Fruit2）
set	set2
show	show2

しかし，実際は，このように名前を変える必要はない．派生クラスで基本クラスのメンバー関数を**オーバーロード**（上書き）できるからである．

5.1◆オーバーロード

関数の**オーバーロード**とは，派生クラスで同じ関数の本体を書き換えて処理内容を変更してしまうことを意味する．ここで**同じ関数**とは，名前が同じだけでなく，引数も同じことを指す．

派生クラスで，基本クラスのメンバー関数の機能を変えたいならオーバーロードすることができる．Fruit2の例では，showやsetの名前を変える必要はなかったのである．

```
class Fruit2 : public Fruit1 {
    int orange;   // orange > 0
public:
    void set(int n1, int n2); ——— 前はset2の名前を使った
    void show(); ————————— 前はshow2の名前を使った
};
```

※この場合のsetはオーバーロードではない（なぜか？）

まずshowから見ていく．showは，メンバー変数の値を表示する．基本クラスFruit1での定義は次のようなものだった．

```
void Fruit1::show()
{
    cout << "apple = " << apple << endl;
}
```

しかし，派生クラスのFruit2ではメンバー変数が増えているので，これでは不足する．そこで，次のように，この定義を書き換える．

```
void Fruit2::show()
```

```
{
    cout << "apple = " << apple << endl;
    cout << "orange = " << orange << endl;
}
```

これが，メンバー関数のオーバーロードである．メンバー関数のオーバーロードでは，関数の形が基本クラスと派生クラスでまったく同じである．すると，どちらのメンバー関数が実行されるかという疑問が残る．

この点は明確である．基本クラスのオブジェクトがshowを呼び出せば，基本クラスのshowが実行される．派生クラスのオブジェクトがshowを呼び出せば，派生クラスのshowが実行される．

```
Fruit1 f1;
f1.show(); ―――――――― これは基本クラスのshowが実行される

Fruit2 f2;
f2.show(); ―――――――― これは派生クラスのshowが実行される
```

5.2◆多重定義

次にメンバー関数のsetを見る．これは，派生クラスにおいて定義を書き換えているのでオーバーロードに見えるが，オーバーロードではない．

setは，メンバー変数に値をセットする．基本クラスのsetは，次のような定義であった．

```
void Fruit1::set(int n)
{
    apple = (n < 0) ? 0 : n;
}
```

一方，派生クラスではメンバー変数が増えているので，setの定義を次のように変えている．

```
void Fruit2::set(int n1, int n2)
{
    apple = (n1 < 0) ? 0 : n1;
    orange = (n2 < 0) ? 0 : n2;
}
```

ここで，setの引数が基本クラスと派生クラスで異なることに注意．名前は同じであるが，引数が異なる．

引数が異なれば，同じ名前の関数を複数個定義することができる．ゆえに，このsetはメンバー関数のオーバーロード（同じ関数を書き換える）でなく，関数の多重定義（引数の異なる同じ名前の関数を複数個定義する）である．

実際，Fruit2は引数の異なる2つのsetを持つことになる．引数を1個指定すればFruit1のsetが実行されるし，引数を2個指定すればFruit2のsetが実行される．

```
Fruit2 f;
f.set(100);            ―― Fruit1のset
f.set(100, 200);       ―― Fruit2のset
```

5.3◆スコープ解決演算子

メンバー関数をオーバーロードしたとき，派生クラスで基本クラスのメンバー関数を使いたいときがある．たとえば，Fruit2のshowがそうである．

```
void Fruit2::show()
{
    cout << "apple = " << apple << endl;    ―― (1)
    cout << "orange = " << orange << endl;
}
```

（1）の部分は，基本クラスのshowを呼び出せば，わざわざ書かなくても済む．この例ではわずか1文なのでそれほどでもないが，（1）の部分が大きいと基本クラスのshowを呼び出したくなる．しかし，ここでshow(); とやっても自分自身のshowを呼び出すだけである．

派生クラスでオーバーロードした基本クラスのメンバー関数を呼び出すときは，関数名の前にスコープ解決演算子を付ける．これは，メンバー関数を定義するときにクラス名を付けるのと同じで，基本クラス名の後ろに2つのセミコロンを付ける．

```
Fruit1::show();
------
  ↑
  これ
```

新しいshowは次のようになる.

```
void Fruit2::show()
{
    Fruit1::show();  ◀───────── ここ
    cout << "orange = " << orange << endl;
}
```

なお，多重定義したsetも，基本クラスのsetを呼び出すときはスコープ解決演算子を付ける.

```
void Fruit2::set(int n1, int n2)
{
    Fruit1::set(n1);  ◀───────── ここ
    orange = (n2 < 0) ? 0 : n2;
}
```

クラスの外から基本クラスのメンバー関数を呼び出すときも同様である.

```
int main()
{
    Fruit2 f;

    f.set(2000, 2500);
    f.show();
    f.Fruit1::set(1000);  ◀───────── ここ
    f.show();

    return 0;
}
```

5.4◆サンプルプログラム

果物クラスをメンバー関数のオーバーロード／多重定義を使って書き換えたものを示す.

【サンプルプログラム】

```
#include <iostream.h>

class Fruit1 {
protected:
    int apple;                  // apple >= 0
public:
    void set(int n);
    void show();
};

void Fruit1::set(int n)
{
    apple = (n < 0) ? 0 : n;
}

void Fruit1::show()
{
    cout << "apple = " << apple << endl;
}

class Fruit2 : public Fruit1 {
    int orange;                 // orange > 0
public:
    void set(int n1, int n2);
    void show();
};

void Fruit2::set(int n1, int n2)
{
    Fruit1::set(n1);
    orange = (n2 < 0) ? 0 : n2;
}

void Fruit2::show()
{
    Fruit1::show();
    cout << "orange = " << orange << endl;
}

int main()
{
    Fruit2 f;
```

```
    f.set(1000, 2300);
    f.show();
    f.Fruit1::set(1500);
    f.show();

    return 0;
}
```

【実行例】

```
% a.out
apple = 1000 ──────────── 基本クラスのshowが実行された
orange = 2300
apple = 1500 ──────────── 派生クラスのshowが実行された
orange = 2300
% _
```

6 継承とコンストラクタ

これまで派生クラスでコンストラクタを使うことを意識的に避けてきた．この節では，継承がコンストラクタに及ぼす影響について説明する．

6.1◆基本クラスのコンストラクタ

簡単な例から始める．基本クラスにはコンストラクタがあって，その派生クラスにはコンストラクタがない場合，どのようになるかという問題である．

まず次が，その基本クラスである．

```
class Base {
public:
    Base() {
        cout << "Base::Base() running ..." << endl;
    }
};
```

話を簡単にするため，このクラスにはコンストラクタしかメンバーが存在しない．しかも，そのコンストラクタはただメッセージを表示するだけなので，インライン関数にしている．

そして，次がその派生クラスである．

```
class Derive : Base {
};
```

これも話を簡単にするため，この派生クラスにはメンバーがない．メンバーを追加していない．したがって，基本クラスと同じメンバーだけを持つことになる．

この状態で，派生クラスのオブジェクトを作るとどうなるか？

```
Derive d; ―――― dは派生クラスDeriveのオブジェクト
```

それを次のサンプルプログラムで確かめる．

【サンプルプログラム】

```
#include <iostream.h>

class Base {
public:
    Base() {
        cout << "Base::Base() running ..." << endl;
    }
};

class Derive : Base {
};

int main()
{
    Derive d; ―――― 派生クラスを使う

    return 0;
}
```

main関数は，ただDerive型の変数を宣言しているだけである．このサンプルプログラムをコンパイルして実行すると次のようになる．

【実行例】

```
% a.out
Base::Base() running ...――――基本クラスのコンストラクタが実行されている
% _
```

これを見ると，基本クラスのコンストラクタが実行されているのがわかる．このように，基本クラスにコンストラクタが定義されていると，派生クラスのオブジェクトを作成したとき，そのコンストラクタが自動的に実行される．

6.2◆派生クラスのコンストラクタ

もちろん派生クラスもコンストラクタを持つことができる．派生クラスのオブジェクトを生成すると，派生クラスのコンストラクタが自動実行される．それだけではなく，基本クラスのコンストラクタも自動実行されるのである．

つまり，派生クラスのオブジェクトを作ると，基本クラスと派生クラスのコンストラクタが実行される．では，どちらのコンストラクタが先に実行されるだろう？　それを次のサンプルプログラムで確かめる．

【サンプルプログラム】

このサンプルプログラムには，3つのクラスがある．Aを基本にA→B→Cのように継承している．つまり，AはCのおじいさんというわけである．各クラスには，それぞれのコンストラクタが定義してある．

この状態でCのオブジェクトを作る．そのとき，どのコンストラクタがどの順序で実行されるかを調べる．

```
#include <iostream.h>

class A {
public:
    A() {
        cout << "A::A() running ..." << endl;
    }
};

class B : A {――――――――――――――――BはAの派生クラス
public:
    B() {
        cout << "B::B() running ..." << endl;
```

```
    }
};

class  C : B { ──────────────Cは Bの派生クラス
public:
    C() {
        cout << "C::C() running ..." << endl;
    }
};

int main()
{
    C c; ──────────────Cのオブジェクトを作る

    return 0;
}
```

【実行例】

```
% a.out
A::A() running ...
B::B() running ...
C::C() running ...
% _
```

これを見ると，すべてのクラスのコンストラクタが実行されているのがわかる．コンストラクタは，次の順序で呼び出される．

基本クラス → 派生クラス

基本クラスがさらに別の基本クラスの派生クラスになっていれば，その基本クラスも実行される．そして，最も古い祖先にさかのぼってコンストラクタが実行される．

基本クラスのコンストラクタが先に実行されるのには理由がある．

派生クラスは，基本クラスのすべてのメンバーを含む．基本クラスのメンバーは，基本クラスのコンストラクタが初期化する．派生クラスのコンストラクタは，残りのメンバーを初期化する．このため，派生クラスのコンストラクタが実行されるときには，すでに基本クラスのコンストラクタが実行されていなければならない．これが，基本クラスのコンストラクタが先に実行される理由である．

6.3◆引数

基本クラスのコンストラクタが引数を持つと，扱いに気をつけなければならない．次のクラスAを例に説明する．

```
class A {
protected:
    int a;
public:
    A(int n) {  ―――― 引数付きのコンストラクタ
        cout << "A::A() running ..." << endl;
        a = n;  ―――― メンバー変数を初期化
    }
};
```

このクラスは，aというメンバー変数を持っている．この変数は，派生クラスから参照される必要があるので，protectedメンバーにしてある．コンストラクタはaを初期化するため，引数をとる．

次に，このクラスの派生クラスBを見る．Bには，メンバー変数が1つ追加されている．

```
class B : public A {
    int b;  ―――― 追加のメンバー変数
};
```

Bのコンストラクタは，2つのメンバー変数a，bを初期化するため，2つの引数をとる．

```
B::B(int n1, int n2)
{
}
```

n2は，Bで追加したbの初期化に使われる．

```
B::B(int n1, int n2)
{
    b = n2;
}
```

問題は，基本クラスのメンバー変数aを初期化するn1である．これは，基本クラスのコンストラクタを呼び出し，そこで初期化する．つまり，n1を基本クラスのコンストラクタに渡す必要がある．

基本クラスのコンストラクタを明示的に呼び出すには，次のようにすればよい．

```
B::B(int n1, int n2)
{
    A::A(n1); ———————————————— (1)
    b = n2;
}
```

しかし，これには大きな問題がある．第6.2節でも説明したように，派生クラスのコンストラクタが実行される前に基本クラスのコンストラクタが実行される．つまり，（1）のコードが実行される前に，すでに基本クラスのコンストラクタは実行されている．そこにさらに（1）を実行すると，コンストラクタが2回実行されてしまう．

さらにいえば，このように書いたプログラムは，コンパイルエラーになる．

図22-4▼

```
go.C: In method `B::B(int, int)':
go.C:17: no matching function for call to `A::A ()'
go.C:11: candidates are: A::A(const A &)
go.C:7:                                  A::A(int)
go.C:17: in base initialization for class `A'
```

これは，基本クラスのコンストラクタを自動的に呼び出そうとしたが，引数が指定されていないのでエラーになっている．基本クラスのコンストラクタに引数を渡すには，どうしたらよいだろう．

6.4◆イニシャライザ

基本クラスのコンストラクタが引数を持つとき，引数を渡すのに特別な構文が用意されている．基本クラスのコンストラクタは，派生クラスのコンストラクタが実行される前に実行される．そこで，基本クラスの呼び出しを関数定義の本体ではなく，その前に書くのである．

```
B::B(int n1, int n2)
{
    A::A(n1);        ← ここに書いてはダメ
    b = n2;
}
```

```
B::B(int n1, int n2) : A(n1)        ← ここに書くのが正解
{
    b = n2;
}
```

あるいは，インライン関数にするなら，次のように書く．

```
class B : public A {
    …………
    B(int n1, int n2) : A(n1) {        ← ここ
    …………
    }
};
```

派生クラスのコンストラクタにおけるこのような書き方を，コンストラクタの**イニシャライザ**という．

【サンプルプログラム】

これは，派生クラスのコンストラクタから基本クラスのコンストラクタへ引数を渡すサンプルプログラムである．

```
#include <iostream.h>

class A {
protected:
    int a;
public:
    A(int n) {
        cout << "A::A() running ..." << endl;
        a = n;
    }
};
```

```
class B : public A {
    int b;
public:
    B(int n1, int n2) : A(n1) {
        b = n2;
        cout << "B::B() running ..." << endl;
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};

int main()
{
    B b(100, 200);

    return 0;
}
```

【実行例】

```
% a.out
A::A() running ...
B::B() running ...
a = 100 ―――――――― 引数が渡されている
b = 200
% _
```

第23章 多態性

「多態」というのは，「多くの態(すがた)を持つ」といった意味である．オブジェクト指向プログラミングでいう多態とは，もちろん「オブジェクトが多態である」ことをいう．

一般にオブジェクト指向プログラミングというと，カプセル化によるデータの隠蔽や継承だけが強調される傾向にある．しかし，この多態性こそがオブジェクト指向プログラミングの威力が発揮される部分である．そのため，多態性はカプセル化や継承に比べると，その概念を理解するのが難しい．おまけにポインタを多用する．

この章では，多態性オブジェクトについて説明する．第3節で，多態性の応用として「仮想GUIプログラミング」を紹介する．ウィンドウのGUI部品がどのように作られているかという興味深いテーマを簡単なプログラムで明らかにする．

1 オブジェクトとポインタ

多態性のポイントは，**オブジェクトの代入互換性**にある．これは，オブジェクトへのポインタに関する規則である．すなわち，多態性は**オブジェクトへのポインタ**と密接な関係がある．

このように，多態性ではポインタを多用する．そこで，この節では，ポインタについての復習を行う．オブジェクトとポインタの関係を再認識する．その後，次の節でオブジェクトの代入互換性について説明する．

1.1◆オブジェクトとポインタ

いまFooというクラスがあるとする．次の“f”はFoo型の変数――すなわち，Foo型のオブジェクトである．

```
Foo f;
```

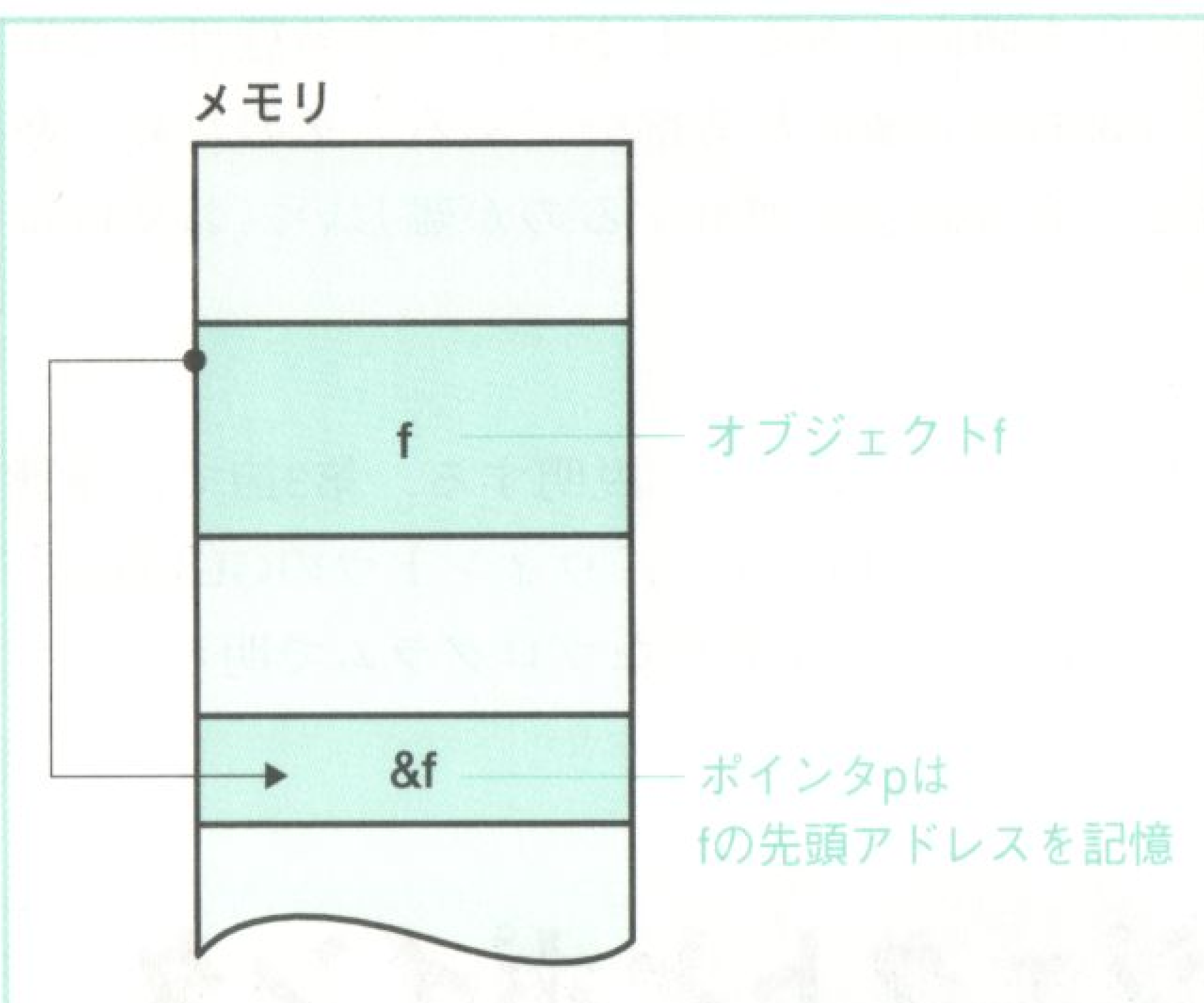
一方，次のように宣言された“p”は，Foo型へのポインタである．

```
Foo *p;
```

この宣言では，オブジェクトが作られないことに注意．pはポインタであって，オブジェクトではない．pはFoo型オブジェクトのアドレスを記憶することができる．たとえば次のようにすれば，pはFoo型のオブジェクトfのアドレスを記憶する．

```
p = &f;
```

図23-1▶



まず，ここまでをしっかりと理解した上で次に進まれたい．

1.2◆ポインタとメンバーの参照

ポインタがオブジェクトを指している場合，そのオブジェクトのメンバーを参照するときは注意が必要である．もし，これが普通のオブジェクト型変数なら，オブジェクト名とピリオドで修飾する．

foo.func(); ———— オブジェクトfooのメンバー関数funcを参照

しかし，ポインタの場合は，（ピリオドではなく）**間接参照演算子** ->を使う．

p = &foo;

p->func();

次に例を示す．

【サンプルプログラム】

これは，間接演算子->の使い方を示すものである．

```
#include <iostream.h>

class A {
public:
    void func() {
        cout << "A::func() running ..." << endl;
    }
};

int main()
{
    A foo;
    A *p;

    p = &foo;
    p->func(); ———— ポインタを通してメンバー関数を参照

    return 0;
}
```

【実行例】

```
% a.out
A::func() running ...
% _
```

2 オブジェクトの代入互換性

多態性は，**オブジェクトの代入互換性**と大いに関係がある．オブジェクトの代入互換性とは，次のことをいう．

オブジェクトへのポインタの代入互換性
基本型のポインタに派生型のポインタを代入できる

この代入互換性自体は，多態性とは直接関係ない．しかし，この代入が可能であるからこそ，多態性が有効に機能する．それをこれから説明していく．

2.1◆代入互換性

代入互換性というのは，次の2つの意味を持っている．

1. 代入演算子（たとえば，=など）で代入できる
2. 関数の引数で値を渡すことができる

要するに，次のように代入できる．

```
pbase = pderive;
```
―― 基本型のポインタに派生型のポインタを代入

あるいは，関数の引数にして値を渡すことができるということである．

```
void add(Base *p)
{
    p->show();
}

add(&Derive);
```
void add(Base *p) ―― Baseへのポインタを受け取る

add(&Derive); ―― &DeriveはBase *と代入互換性がなければならない

で，オブジェクトにおける代入互換性の規則によると，基本型へのオブジェクトに派生型へのオブジェクトを代入できるというのである．次にその例を示す．

2.2◆オブジェクトの代入互換性

オブジェクトの代入互換性について具体例を示す．そのためには，基本型とそこから派生した型が必要である．そこで，次の基本型を用意する．

```
class Base {
public:
    void show() {
        cout << "Base::show() running ..." << endl;
    }
};
```

このクラスは，showというメンバー関数だけを持つ．showは，自分の関数名を表示するだけの関数である．

続いてBaseの派生型Deriveを用意する．

```
class Derive : public Base {
public:
    void show() {
        cout << "Derive::show() running ..." << endl;
    }
};
```

この派生クラスもshowというメンバー関数だけを持つ（オーバーロードしている）．派生クラスのshowであることがわかるように，表示の内容を変えている．

次に，これらのクラスのオブジェクトを作る．

```
Base b;  ———————— bは基本型のオブジェクト
Derive d;  ———————— dは派生型のオブジェクト
```

これで準備が整った．オブジェクトの代入互換性をテストすることができる．オブジェクトの代入互換性によると，次のような代入が可能であるという．

```
Base *p = &d;  ———————————— (1)
```

ここでpは，Base型——すなわち，基本型へのポインタである．ゆえに，基本型のオブジェクトを代入することは当然可能である．

```
p = &b    ――――――――― この代入は当然OK
```

しかし，オブジェクトの代入互換性によると，pに派生型オブジェクトへのポインタを代入してもよいというのである．すなわち（1）のように代入してもエラーにはならない．

しかし，その逆はダメである．

```
Derive *p = &b;    ――――――― 派生型のポインタに
                            基本型のポインタを代入するのはダメ
```

このようなことをすると，コンパイルエラーになる．

```
type `Derive' is not a base type for type `Base'
```

これが代入互換性である．

2.3◆基本型への回帰

一般に基本型のポインタは，そのポインタが指すオブジェクトが派生型であったとしても基本型と見なす．派生型で追加されたメンバーを使うことはできない．また，派生型でオーバーロードした関数を呼び出しても，上書きされる前の基本型の関数が実行される．

たとえば，次のdは，派生型のオブジェクトである．

```
Derive d;    ――――――――― dは派生型のオブジェクト
```

そのdのアドレスを基本型のポインタに代入する．これは，代入互換性により許される．

```
Base *p = &d;
```

pは派生型のオブジェクトdを指す．しかし，pを通しての操作は，それを基本型のオブジェクトと見なす．たとえば，次のようにすると，このshowは，派生型のshowではなく，基本型のshowを実行する．

```
p->show();
```

それを次のサンプルプログラムで確かめる．

【サンプルプログラム】

```
#include <iostream.h>

class Base {
public:
    void show() {
        cout << "Base::show() running ..." << endl;
    }
};

class Derive : public Base {
public:
    void show() {
        cout << "Derive::show() running ..." << endl;
    }
};

int main()
{
    Base b;
    Derive d;

    Base *p = &b;
    p->show();  ―――――― pは基本型オブジェクトを指している
    p = &d;
    p->show();  ―――――― pは派生型オブジェクトを指している

    return 0;
}
```

関数mainに注目する．ここでは，最初に基本型のオブジェクトと派生型のオブジェクトを作成している．

```
Base b;
Derive d;
```

次に基本型へのポインタを作り，そこに基本型オブジェクトを代入している．

```
Base *p = &b;
```

そして，showを実行している．このshowは，当然，基本型のshowである．

```
p->show(); ―――――― これは基本型のshow
```

次に派生型オブジェクトのアドレスを代入し，やはりshowを実行している．問題は，このshowが基本型のものになるか，はたまた派生型のものになるかということである．

```
p = &d;        派生型のオブジェクトを代入
p->show();
```

実行例は，次のようになる．

【実行例】

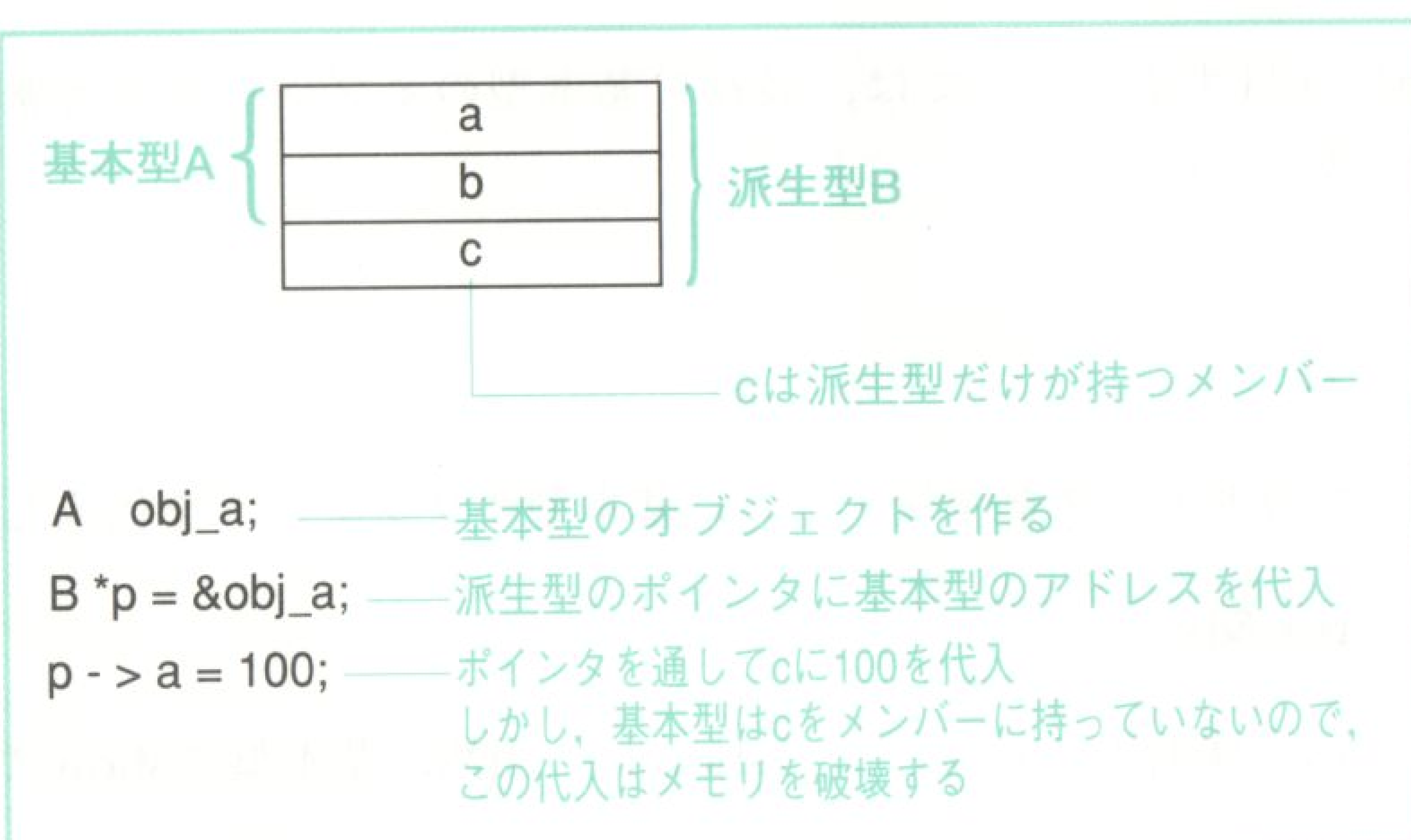
```
% a.out
Base::show() running ...
Base::show() running ...
% _
```

いずれも基本型のshowが実行されているのがわかる．これが「基本型への回帰」である．

基本型のポインタに派生型のポインタを代入可能なのは，基本型が派生型のサブセットになっているからである．派生型は，基本型にメンバーを追加したものである．派生型で追加されたメンバーを切り捨てることによって，派生型を基本型として扱うことができる．

逆に「基本型を派生型に代入する」ことを許すと，とんでもないことになる．基本型は，派生型で追加されたメンバーを持っていない．したがって，それらのメンバーを参照したり，メンバーpが基本型オブジェクトを指している関数を呼び出したりすると，とんでもないことになってしまう．

図23-2▶



基本型のポインタに代入された派生型のオブジェクトは，基本型に回帰する——この点は重要である．その規則を破るのが仮想関数であり，それが多態性へとつながる．この点については，再度第6節で取り上げる．

2.4◆派生型への回帰

もし，基本型のポインタに代入された派生型のオブジェクトを派生型として扱いたいなら，派生型へのポインタにキャストすればよい．書き方は，次のようになる．

```
((Derive *) p)->show();
```

ちょっとややこしい表現になっているので分解すると，まずpは基本型へのポインタである．

```
p ──────────── これは基本型Baseへのポインタ
```

これを派生型へのポインタと見なすには，次のようにキャストする．

```
(Derive *) p ──────────── これは派生型Deriveへのポインタ
```

このポインタを通してshowを実行するには，->show() を付ける．

```
(Derive *) p->show();
```

ただし，これではp->show()をキャストすることになってしまうので，(Derive *) p全体を（）で囲んで優先順位を変える．

```
((Derive *) p)->show();
```

これで出来上がりである．これを試すためのサンプルプログラムを次に示す．

【サンプルプログラム】

これは，基本型へのポインタを派生型へのポインタにキャストする例である．前項のサンプルプログラムのmainを次のように書き換える．

```
int main()
{
    Base b;
    Derive d;
```

```
    Base *p = &d;               基本型のポインタに派生型を代入
    ((Derive *) p)->show();     派生型として実行

    return 0;
}
```

【実行例】

```
% a.out
Derive::show() running ...     派生型のshowが実行されている
% _
```

2.5◆引数の代入互換性

オブジェクトの代入互換性は，代入のみならず，関数の引数についてもいえる．たとえば，次のような関数addを例に説明する．

```
void add(Base *p)
{
    p->show();
}
```

この関数は，基本型Baseへのポインタを引数にとる．この関数に対して，派生型オブジェクトのアドレスを渡すことができる．

```
Derive d;     dは派生型のオブジェクト
add(&d);      dのアドレスを渡す
```

これも基本型へのポインタに派生型を代入できるという代入互換性によるものである．関数addの中では，pは（たとえ派生型オブジェクトのアドレスが渡されても）基本型へのアドレスとして処理される．

以上を，次のサンプルプログラムでテストする．

【サンプルプログラム】

```
#include <iostream.h>

class Base {
```

```
public:
    void show() {
        cout << "Base::show() running ..." << endl;
    }
};

class Derive : public Base {
public:
    void show() {
        cout << "Derive::show() running ..." << endl;
    }
};

void add(Base *p)
{
    p->show();
}

int main()
{
    Base b;
    Derive d;

    add(&b);        // 基本型のオブジェクトを渡す
    add(&d);        // 派生型のオブジェクトを渡す

    return 0;
}
```

関数mainでは，基本型のオブジェクトbと派生型のオブジェクトdを作成し，それぞれのアドレスを関数addに渡している．関数addの中では，渡されたオブジェクトのshowを呼び出している．しかし，いずれのオブジェクトも基本型と見なされるので，実行されるshowは基本型のshowになる．

次に実行例を示す．

【実行例】

```
% a.out
Base::show() running ...     基本型のオブジェクトを渡した
Base::show() running ...     派生型のオブジェクトを渡した
%
```

3 仮想GUIプログラミング

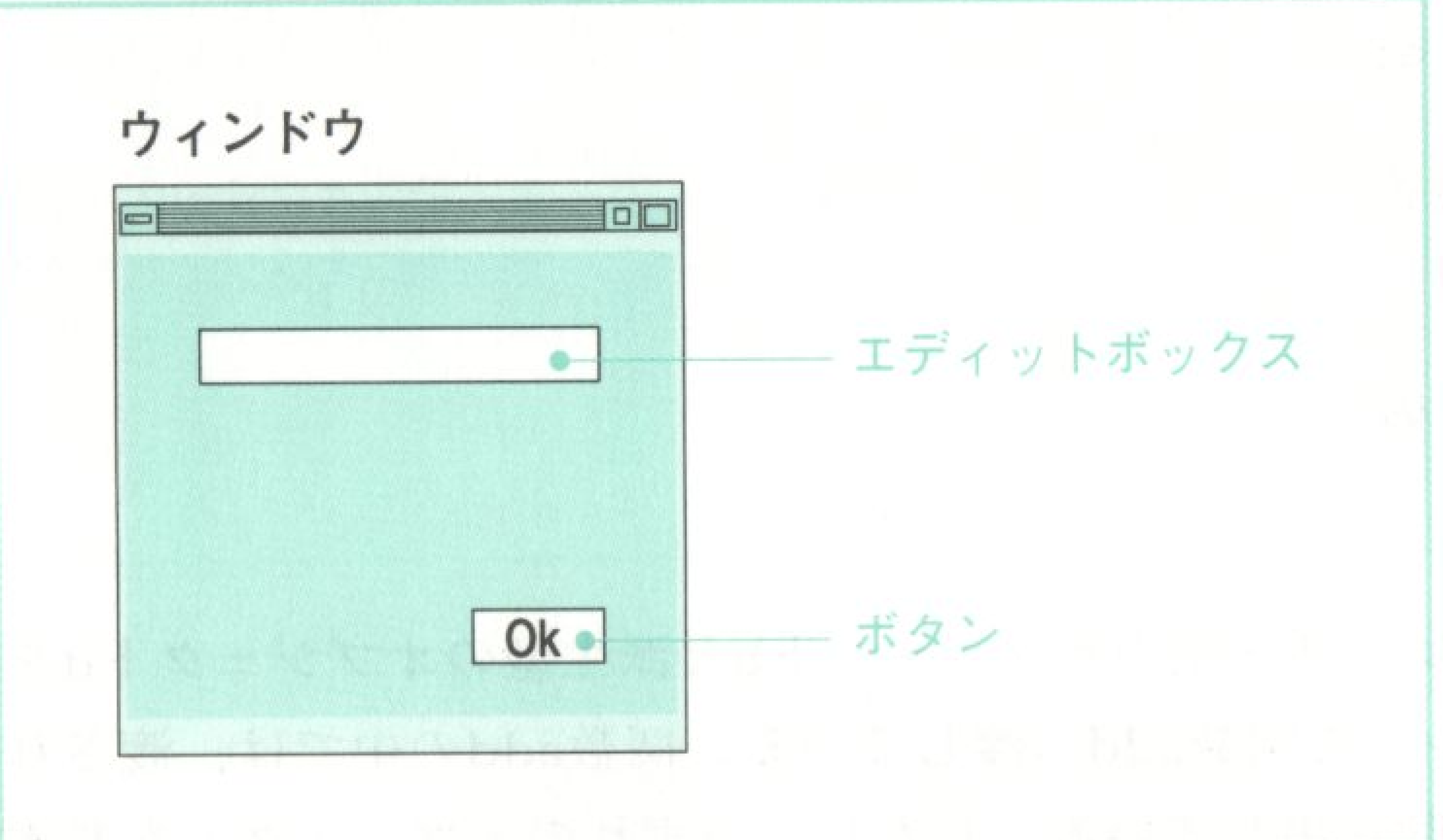
「基本型のポインタには派生型のポインタを代入できる（オブジェクトの代入互換性）」ことを説明してきた．これが何の役に立つのかというと，これにより「基本型のオブジェクトを使って，そこから派生するすべての派生型を管理する」ことが可能になる．

そこで，そのことを説明するため，この節では仮想的なGUI（グラフィカルユーザーインターフェース）のプログラミングに取り組む．

3.1◆GUI

GUIというと，Xなどのウィンドウシステムでボタンやエディットボックスなどを配置してユーザーとの対話を進めるプログラムを思い浮かべるだろう．この節では，ボタンとエディットボックスを持つ簡単なプログラムを作成する．

図23-3▶



ここで作成するプログラムは，GUIの部品とオブジェクト指向プログラミングとの関係を明らかにするのが目的である．あくまでもGUIのプログラムがどのような仕組みで作られているかを示すのが目的なので，それほど凝ったことはやらない．

ボタンといってもグラフィカルなボタンではない．ただ，［Button : Ok］と表示するだけの仮想的なものである．エディットボックスといってもテキストの編集ができるわけではない．ただ［Edit :電話番号］と表示するだけの仮想的なものである．

3.2◆GUI部品のためのクラス

GUIのプログラミングでは，ボタンやエディットボックスのようなGUIのボタンを1つのオブジェクトで表現する．つまり，ボタンやエディットボックスのためのクラスを作る．

```
class Button { ――――――ボタンのためのクラス
    ............
};

class Edit { ――――――エディットボックスのためのクラス
    ............
};
```

ただし，普通は，このようにそれぞれのGUI部品を関連のない別のクラスにすることはしない．GUIの部品に共通する性質を抽出したクラスを作り，それぞれの部品はそこから派生して作る．

```
class View { ――――――すべてのGUI部品の基本クラス
    ............
};
```

ボタンやエディットボックスは，このViewクラスの派生クラスにする．

```
class Button : public View { ――――ボタンのためのクラス
    ............
};

class Edit : public View { ――――エディットボックスのためのクラス
    ............
};
```

3.3◆オブジェクト階層図

このように，すべてのGUI部品を1つのViewクラスから派生して作ると，次のようなオブジェクトの階層図が出来上がる．

```
View
 ├──────── Button
 └──────── Edit
```

オブジェクトの代入互換性によると，ButtonやEditへのポインタをViewへのポインタに代入することができる．つまり，View型のポインタを用意すれば，そのポインタを使ってボタンやエディットボックスなどすべてのGUI部品を管理することができる．これが，オブジェクト指向プログラミングによるGUIプログラミングの考え方である．

3.4◆GUI部品の基本クラス

すべてのGUI部品は，たとえば次のような情報が必要である．

```
(x, y) ──────────── その部品の表示位置
name ───────────── その部品の名前
show() ──────────── その部品を表示するためのメンバー関数
```

したがって，すべてのGUI部品の基本クラスViewは，次のような感じになる．

```
class View {
protected:
      int homex; ──────── その部品のx座標
      int homey; ──────── その部品のy座標
      char name[20]; ──── その部品の名前
public:
      show();
};
```

ただし，この節で作成するプログラムは，それほど凝ったことはしない．実際にGUI部品を表示するわけではないので座標は使わない．次のような簡単なViewクラスで話を進める．

```
class View {
protected:
      char name[20]; ──── その部品の名前
public:
```

```
    View(char *s);  ———— コンストラクタ
    show();  ———— その部品を表示する
};
```

●name

メンバー変数のnameは，その部品の名前を登録する．この変数の決められた用途はない．プログラマーが自由に使うことができる．

一例を挙げると，部品ごとに一意の名前を付けてname変数に保存する．すると，name変数で部品を識別することができる．たとえば3つのエディットボックスを用意したとき，“Edit1”“Edit2”“Edit3”のような名前を付ける．そうすれば，name変数でそれぞれのエディットボックスを識別することができる．

●コンストラクタ

メンバー変数nameに初期値を与えるため，コンストラクタを用意する．コンストラクタのコードは，次のようになる．

```
View::View(char *s)
{
    strcpy(name, s);
}
```

●show

メンバー関数のshowは，その部品を表示する．ただし，Viewクラスは仮想的なクラスなので，その中身は空である．

```
void View::show()
{
}
```

実際の表示は，各部品クラスのshowで行う．

3.5◆ボタンクラス

Viewクラスを派生してボタンクラスを作る．

```
class Button : public View {
```

```
public:
    Button(char *s);
    void show();
};
```

コンストラクタのButtonは，基本クラスのViewのコンストラクタを呼び出してnameの初期化を行う．基本クラスのコンストラクタの呼び出しには，第22章の「**継承とコンストラクタ**」のところで説明したイニシャライザを使う．

イニシャライザ

```
Button::Button(char *s) : View(s)
{
}
```

メンバー関数のshowを，ボタンを表示するようにオーバーロードする．

```
void Button::show()
{
    cout << "Button : " << name << endl;
}
```

これでボタンクラスの完成である．ここまでをテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

これは，ボタンクラスを使ってOkボタンとCancelボタンを作り，表示する．

```
#include <iostream.h>
#include <string.h>

class View {                    すべての部品クラスの基本クラス
protected:
    char name[20];              部品の名前
public:
    View(char *s);
    void show();
};

View::View(char *s)
{
```

```
    strcpy(name, s);
}

void View::show()
{
}

class Button : public View { ――――――――― ボタンクラス
public:
    Button(char *s);
    void show();
};

Button::Button(char *s) : View(s)
{
}

void Button::show()
{
    cout << "Button : " << name << endl;
}

int main()
{
    Button b1("Ok"); ―――――――――――――― Okボタン
    Button b2("Cancel"); ――――――――――― Cancelボタン

    b1.show();
    b2.show();

    return 0;
}
```

【実行例】

```
% a.out
Button : Ok
Button : Cancel
% _
```

3.6◆エディットボックスクラス

同じようにして，Viewクラスを派生してエディットボックスクラスを作ること

ができる．

```
class Edit : public View {
public:
    Edit(char *s) : View(s) {}
    void show() {
        cout << "Edit : " << name << endl;
    }
};
```

エディットボックスクラスは，簡単のため，すべての関数をインライン関数にしている．

【サンプルプログラム】

これは，エディットボックスクラスを使って3つのエディットボックスを作っている．それぞれのエディットボックスには“名前”“住所”“電話番号”の名前を与えてある．

```
#include <iostream.h>
#include <string.h>

class View {                         ――― すべての部品クラスの基本クラス
protected:
    char name[20];                   ――― 部品の名前
public:
    View(char *s);
    void show();
};

View::View(char *s)
{
    strcpy(name, s);
}

void View::show()
{
}

class Button : public View {         ――― ボタンクラス
public:
    Button(char *s);
```

```
    void show();
};

Button::Button(char *s) : View(s)
{
}

void Button::show()
{
    cout << "Button : " << name << endl;
}

class Edit : public View {         エディットボックスクラス
public:
    Edit(char *s) : View(s) {}
    void show() {
        cout << "Edit : " << name << endl;
    }
};

int main()
{
    Edit e1("名前");
    Edit e2("住所");
    Edit e3("電話番号");

    e1.show();
    e2.show();
    e3.show();

    return 0;
}
```

【実行例】

```
% a.out
Edit :名前
Edit :住所
Edit :電話番号
% _
```

3.7◆「基本型への回帰」の弊害

第3.5節，第3.6節のサンプルプログラムは，ボタンクラス，エディットボックスそれぞれのオブジェクトを作成し，**単独に**利用するものであった．これでは，多態性に進むことはできない．

オブジェクトを階層的に構成し，基本型のオブジェクトを使ってそこから派生するすべての派生型を管理するには，基本型Viewへのポインタを用意し，そのポインタを通してオブジェクトを操作する．これが多態性を理解する第一歩になる．

そこで，まず次のようにView型へのポインタvpを宣言する．

```
View *vp;
```

そして，ボタンオブジェクトとエディットボックスオブジェクトを1つずつ作成する．

```
Button b("Ok");———————bはボタンオブジェクト
Edit e("名前");———————eはエディットボックスオブジェクト
```

最初は，vpにボタンのアドレスをセットし，showを呼び出す．

```
vp = &b;
vp->show(); ———————ボタンが表示されるはずだが？
```

これで，ボタンが表示されるだろうか？

もう1つ——．今度は，vpにエディットボックスのアドレスをセットし，showを呼び出す．

```
vp = &e;
vp->show(); ———————エディットボックスが表示されるはずだが？
```

今度は，エディットボックスが表示されるだろうか？

もちろん，正しい答えは「何も表示されない」である．第2.3節「**基本型への回帰**」を十分に理解していればわかるように，vpは基本型Viewへのポインタである．たとえ，ここに派生型へのアドレスを代入しても基本型と見なされる．したがって，vp->show(); で実行されるshowは，Viewクラスのshowである．Viewクラスのshowは空っぽである（第3.4節を参照）．これを実行しても何も表示されない．

以上を，次のサンプルプログラムでテストする．

【サンプルプログラム】

```
#include <iostream.h>
#include <string.h>

class View {              ―――― すべての部品クラスの基本クラス
protected:
    char name[20];        ―――― 部品の名前
public:
    View(char *s);
    void show();
};

View::View(char *s)
{
    strcpy(name, s);
}

void View::show()
{
}

class Button : public View {   ―――― ボタンクラス
public:
    Button(char *s);
    show();
};

Button::Button(char *s) : View(s)
{
}

void Button::show()
{
    cout << "Button : " << name << endl;
}

class Edit : public View {     ―――― エディットボックスクラス
public:
    Edit(char *s) : View(s) {}
    void show() {
        cout << "Edit : " << name << endl;
```

```
    }
};

int main()
{
    View *vp;

    Button b("Ok");
    Edit e("名前");

    vp = &b;                    ボタンを代入
    vp->show();
    vp = &e;                    エディットボックスを代入
    vp->show();

    return 0;
}
```

【実行例】

```
% a.out
% _                 何も表示されない
```

この何も表示されない状況を検討することは重要である．この状況を改善する努力の中から仮想関数／多態性が生まれる．次の節では，仮想関数／多態性によらない解決を図る．

4 列挙型

本題に入る前に**列挙型**という新しいデータ型の話をする．列挙型は，ユーザー定義型の1つである．

4.1◆enum

列挙型を使うと，複数の定数をまとめて宣言することができる．列挙型を宣言するには，キーワードenumに続けて{}の中に定数名を並べる．

```
enum {cons1, cons2, cons3};
```

この宣言は，次の3つの宣言と等価である．

```
const cons1 = 0;
const cons2 = 1;
const cons3 = 2;
```

ここで，cons1，cons2，cons3は列挙型で定義された定数なので，**列挙定数**と呼ぶことにする．列挙定数は，先頭から0，1，2，… の値が割り当てられる．

4.2◆値の変更

列挙定数に0，1，2，… 以外の値を割り当てることもできる．

```
enum {cons1, cons2, cons3 = 100, cons4};
```

（cons3 = 100 の部分に矢印：注目）

ここで，cons3に100を代入していることに注意．これは次の宣言と同じである．

```
const cons1 = 0;
const cons2 = 1;
const cons3 = 100;
const cons4 = 101;
```

すなわち，代入を行った列挙定数から値が変わり，そこから順に+1された値が割り当てられる．

4.3◆新しいデータ型

列挙型は，ユーザー定義型の1つである．ということは，列挙型が新しい型名を作ることを意味する．列挙型を宣言するとき，次のように {} の前に型名を指定することができる．

```
enum weekday {monday, tuesday, wednesday, thursday, friday};
```

（weekday の部分に矢印：新しい型名になる）

この結果，weekdayが新しいデータ型の名前になる．この結果，次のように

weekday型の変数を宣言できる

```
weekday w;
```

このwには，weekdayの列挙定数を代入できる．

```
w = tuesday;
```

tuesday = 1なので，この代入は（値的には）次と同じである．

```
w = 1;
```

ただし，このように書くと，次のような警告エラーになる．

```
warning: conversion from `int' to `enum weekday'
```

このように，列挙型変数を使うと，不正な代入を見つけることができる．

4.4◆書式について

最後に，enumの宣言における書式に触れておく．

```
enum weekday {monday, tuesday, wednesday, thursday, friday};
```

この宣言は長いので，次のように書いてもよい．

```
enum weekday {
        monday,
        tuesday,
        wednesday,
        thursday,
        friday
};
```

特に，途中に代入が入るときは，この書き方がわかりやすい．

5 非多態性による解決

基本型を祖先とするすべての派生型を基本型へのポインタで管理しようとするとき，「基本型への回帰」が邪魔をする．この節では，多態性を使わずにこの問題を回避する方法を説明する．

5.1◆型情報を示す引数

第3節で起こった問題は，メンバー関数のshowを実行するとき，基本型のshowが実行されてしまうということであった．

ここでやりたいことは，基本型へのポインタvpにボタンオブジェクトが代入されていればボタンクラスのshowを実行したいし，エディットボックスクラスのオブジェクトが代入されていればエディットボックスクラスのshowを実行したいということである．すなわち，vpにどのようなオブジェクトが代入されていても，適切なshowが実行されるようにしたい．

それを解決する1つの方法は，showに引数を追加し，どの型のオブジェクトを対象にしているかを指定できるようにすることである．

まずオブジェクトの種類を示すための列挙定数を用意する．

```
enum objtype {  ——— 部品の種類を示す型
	button, edit
};
```

そして，基本クラスViewのshowにobjtype型の引数を追加する．

```
class View {
	............
	show(objtype type);  ◀——— ここ
};
```

このようにすれば，showは引数のtypeによって，どの部品を表示するかを決めることができる．改良版showのコードは次のようになる．

```
void View::show(objtype type)
```

```
    {
        switch(type) {          ── typeにより処理を分ける
        case button:
            cout << "Button : " << name << endl;
            break;
        case edit:
            cout << "Edit : " << name << endl;
        }
    }
```

showを呼び出すときは，どの部品を表示したいかを引数で知らせるようにする．

```
    vp->show(button);   ── ボタンを表示する
    vp->show(edit);     ── エディットボックスを表示する
```

5.2◆なぜよくないか？

この改良版showを使って書き換えたサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>
#include <string.h>

enum objtype {                  // 部品の種類
    button, edit
};

class View {
protected:
    char name[20];              // 部品の名前
public:
    View(char *s);
    void show(objtype type);
};

View::View(char *s)
{
    strcpy(name, s);
}

void View::show(objtype type)
```

```
{
    switch(type) {
    case button:
        cout << "Button : " << name << endl;
        break;
    case edit:
        cout << "Edit : " << name << endl;
  }
}

class Button : public View {
public:
    Button(char *s);
    void show();
};

Button::Button(char *s) : View(s)
{
}

void Button::show()
{
    cout << "Button : " << name << endl;
}

class Edit : public View {
public:
    Edit(char *s) : View(s) {}
    void show() {
        cout << "Edit : " << name << endl;
    }
};

int main()
{
    View *vp;

    Button b("Ok");
    Edit e("名前");

    vp = &b;
    vp->show(button); ―――――――――― ボタンを表示
    vp = &e;
    vp->show(edit); ――――――――――― エディットボックスを表示

    return 0;
}
```

【実行例】

```
% a.out
Button : Ok
Edit :名前
% _
```

今度はGUI部品が表示されている．このように型情報を示す引数を追加することによって，「基本型への回帰」問題を解決することができる．

しかし，この方法はあまり芳しいものではない．新しい部品を作るたびにshowを書き換えなければならない．第一，基本クラスを継承した意味がほとんどない．派生クラスでオーバーロードしたshowは，ほとんど使われることなく無駄なメモリを費すだけである．

オブジェクト指向プログラミングによる正しい解決法は，**仮想関数**を使うことである．仮想関数を導入することによって，この状況は改善される．いよいよ仮想関数を理解する時がやって来た．

6 仮想関数

いよいよ仮想関数を理解する時がやって来た．**仮想関数**というのは，virtualというキーワードを付けて宣言されたメンバー関数のことである．その仮想関数を持つオブジェクトが**多態性オブジェクト**である．

6.1◆virtualキーワード

仮想関数というのは，virtualというキーワードを付けて宣言されたメンバー関数のことである．以下に，一般的なメンバー関数と，仮想関数を並べて示す．

```
class View {
    ............
    show();  ◀―― これは普通のメンバー関数
};
```

```
class View {
    ............
    virtual show();  ← これは"virtual"が付いているので
};                     仮想関数
```

virtualを付ける位置は宣言のところだけで，定義のところに付ける必要はない．

```
class View {
    ............
    virtual show();  ← ここにはvirtualを付ける
};

void View::show()  ← ここにはvirtualを付けない
{
}
```

もちろん，仮想関数をインライン関数として定義する場合はvirtualを付ける．

```
class View {
    ............
    virtual show() {}  ← 空のインライン関数
};
```

6.2◆仮想関数のオーバーロード

メンバー関数を仮想関数にするとどうなるか？——その関数を普通に使うかぎりはどうにもならない．普通のメンバー関数として振る舞うだけである．

仮想関数が威力を発揮するのは，その関数が派生型でオーバーロードされたときである．たとえば次は，Buttonクラスにおいて仮想関数のshowをオーバーロードしている．

```
class View {
    ............
    virtual show();  ——— これはvirtualが付いているので
};                       仮想関数

class Button : public View { ——— Buttonは派生クラス
```

```
public:
    ............
    show(); ―――――― 仮想関数showをオーバーロード
};
```

なお，仮想関数をオーバーロードするときには，virtualキーワードを付ける必要がない．基本クラスでvirtualが指定されたメンバー関数は，派生クラスでも自動的にvirtualが指定されるからである．

6.3◆「基本型への回帰」の回避

仮想関数がオーバーロードされるとどうなるか？――基本クラスへのポインタを使って仮想関数を実行したとき，適切なバージョンの関数が実行される．そのポインタが派生クラスのオブジェクトを指していれば，その派生クラスの仮想関数が実行される．すなわち「基本型への回帰」が回避されるのである（ややこしい表現！）．

例を挙げる．

```
View *vp; ―――――――― vpは基本型へのポインタ
Button b("Ok"); ――――― bは派生型のオブジェクト
vp = &b; ――――――――― vpは派生型bを指す
vp->show(); ――――――― このshowは，どのバージョンの
                       showが実行されるか？
```

いまvpは派生型のオブジェクトbを指している．このvpを通してshowを実行しようとしている．もしshowが仮想関数でなかったら，第3.7節でも問題になったように基本型のshowが実行されてしまう．しかし，showを仮想関数にしてあれば，適切なバージョン――すなわち，この場合はボタンクラスのshowが実行される．

前の節では，「基本型への回帰」を回避するため，showに余計な引数を追加した．しかし，仮想関数を使えば，この問題を回避することができるのである．

7 多態性

さらに例を続ける．

```
View *vp;

Button b("Ok");
Edit e("名前");

vp = &b;
vp->show(); ――――――――――――― (1)
vp = &e;
vp->show(); ――――――――――――― (2)
```

この例で（1）と（2）はまったく同じ形をしている．しかし，実行されるshowは異なる．（1）はボタンクラスのshowが実行されるし，（2）はエディットボックスクラスのshowが実行される．

このようにvpは，時間とともに姿が変わる．あるときはボタンとして振る舞い，あるときはエディットクラスとして振る舞う．これが，多態性である．

第4節では，「基本型への回帰」を型情報を追加することで回避した．しかし，多態性を使えば，型情報を不要にする．オブジェクト自身が適切な振る舞いをしてくれるからである．

【サンプルプログラム】

これは，多態性をテストするためのサンプルプログラムである．異なる型の複数の派生型を管理するため，基本型のポインタを用意する．

```
const int views = 5; ―――― 配列のサイズ
View *vp[views]; ―――――― 基本型のポインタを要素とする配列
```

それぞれの配列要素にボタンオブジェクトやエディットボックスオブジェクトのアドレスを代入する．そして，forループで一挙にそれらを表示する．

```
for (int i = 0; i < views; i++)
    vp[i]->show();
```

いずれもvp[i]->show();を実行するだけであるが，適切なshowが実行されて，目

的のGUI部品が表示される．

```
#include <iostream.h>
#include <string.h>

class View {
protected:
    char name[20];
public:
    View(char *s);
    void virtual show();          ―――― 仮想関数
};

View::View(char *s)
{
    strcpy(name, s);
}

void View::show()                 ―――― ここにはvirtualを付けない
{
}

class Button : public View {
public:
    Button(char *s);
    void show();
};

Button::Button(char *s) : View(s)
{
}

void Button::show()
{
    cout << "Button : " << name << endl;
}

class Edit : public View {
public:
    Edit(char *s) : View(s) {}
    void show() {
        cout << "Edit : " << name << endl;
    }
};
```

```
int main()
{
    const int views = 5;

    Edit e1("名前");
    Edit e2("住所");
    Edit e3("電話番号");

    Button b1("Ok");
    Button b2("Cancel");

    View *vp[views];

    vp[0] = &e1; ──────────── 基本型のポインタに
    vp[1] = &e2;               いろいろな派生型を代入
    vp[2] = &e3;
    vp[3] = &b1;
    vp[4] = &b2;

    for (int i = 0; i < views; i++)
        vp[i]->show();

    return 0;
}
```

【実行例】

```
% a.out
Edit :名前 ──────────── エディットボックス
Edit :住所
Edit :電話番号
Button : Ok ──────────── ボタン
Button : Cancel
% _
```

8　純粋仮想関数と抽象クラス

ここで，再びすべてのGUI部品の基本クラスとなるViewを見る．

```
class View {
```

```
protected:
    char name[20];  ―――――― その部品の名前
public:
    View(char *s);  ―――――― コンストラクタ
    virtual show(); ―――――― その部品を表示する
};
```

この中で注目したいのは，メンバー関数のshowである．showは，その部品を表示する．しかし，基本クラスそのものを表示することはないので，showの本体は空である．

```
void View::show()
{ ―――――――――――――――――― 中身は空っぽ
}
```

具体的なshowの定義は，それぞれの派生クラスの中で行われる．それゆえに，showは仮想関数として定義されたのであり，Viewクラスのshowは関数の雛形（引数や型）に過ぎない．その中身は必要ない．

このように実体のない仮想関数を宣言するときは，（ただの仮想関数ではなく）**純粋仮想関数**にするとよい．

8.1◆純粋仮想関数

純粋仮想関数を宣言するときは，関数宣言の最後のところに“= 0”を付ける．

```
virtual show() = 0;
```

仮想関数を純粋仮想関数にしたときは，その本体を定義する必要がない．つまり，わざわざ空の関数を定義するという無駄なことをしなくてよい．結果として，わずかではあるが，実行ファイルのサイズも小さくすることができる．

たとえば第6節のサンプルプログラムで，Viewクラスのshowを純粋仮想関数にすると，実行ファイルのサイズが次のように小さくなる

関数の種類	実行ファイルのサイズ
空のshowを使った場合	18366バイト
純粋仮想関数にした場合	18292バイト

【サンプルプログラム】

これは，第6節のサンプルプログラムを純粋仮想関数を使って書き換えたものである．

```
#include <iostream.h>
#include <string.h>

class View {
protected:
    char name[20];
public:
    View(char *s);
    void virtual show() = 0; ――――――― 純粋仮想関数
};

View::View(char *s)
{
    strcpy(name, s);
}

class Button : public View {
public:
    Button(char *s);
    void show();
};

Button::Button(char *s) : View(s)
{
}

void Button::show()
{
    cout << "Button : " << name << endl;
}

class Edit : public View {
public:
    Edit(char *s) : View(s) {}
    void show() {
        cout << "Edit : " << name << endl;
    }
};
```

```
int main()
{
    const int views = 5;

    Edit e1("名前");
    Edit e2("住所");
    Edit e3("電話番号");

    Button b1("Ok");
    Button b2("Cancel");

    View *vp[views];

    vp[0] = &e1;
    vp[1] = &e2;
    vp[2] = &e3;
    vp[3] = &b1;
    vp[4] = &b2;

    for (int i = 0; i < views; i++)
        vp[i]->show();

    return 0;
}
```

【実行例】

```
% a.out
Edit :名前 ──────────── エディットボックス
Edit :住所
Edit :電話番号
Button : Ok ──────────── ボタン
Button : Cancel
% _
```

8.2◆抽象クラス

メンバーに純粋仮想関数を持つクラスを**抽象クラス**という.

抽象クラスは，一種の不完全なクラスである．なぜなら純粋仮想関数という実体のないメンバー関数を持っているからである．そのため抽象クラスのオブジェクトを作ることはできない．たとえば純粋仮想関数のshowを持つViewクラスのオ

ブジェクトを作ろうとすると，次のようなコンパイルエラーになる．

図23-4▼

```
% g++ go.C
go.C: In function `int main()':
go.C:43: cannot declare variable `v' to be of type `View'
go.C:43:   since the following virtual functions are abstract:
go.C:43:         void View::show()
% _
```

そもそも抽象クラスは，そのクラスを直接使うことを想定していない．その派生クラスを統一的に管理するために存在する．したがって，そのクラスのオブジェクトを作ることは誤りである．

純粋仮想関数をメンバーにすることにより，そのようなクラスを抽象クラスにすることができる．そして，誤ってそのクラスのオブジェクトを作ってしまうことを回避することができる．そのためにも純粋仮想関数は威力を発揮する．

8.3◆抽象クラスの派生クラス

抽象クラスの派生クラスは，そのクラスで純粋仮想関数を実体化（関数の本体を定義する）しないかぎり，依然として抽象クラスのままである．

たとえば次のサンプルプログラムは，基本クラスAから派生クラスBAを作っている．基本クラスAはfooという純粋仮想関数を持っているので抽象クラスである．それを派生したBAでもfooを再定義していない．したがってBAも抽象クラスである．

基本クラスA
　　純粋仮想関数fooを持つ　――――　Aは抽象クラス
　　↑
派生クラスBA
　　純粋仮想関数fooに手をつけていない
　　　　　　　　　　　　　――――　BAはやはり抽象クラス

したがって，クラスBAのオブジェクトを作ろうとするとコンパイルエラーになる．

【サンプルプログラム】

これは，上記をテストするためのプログラムである．関数mainでクラスBAのオブジェクトを宣言しようとしているが，BAは抽象クラスなので不可能である．

```
#include <iostream.h>
#include <string.h>

class A {
public:
    virtual foo() = 0;  ——— 純粋仮想関数
};

class BA : public A {
public:
    void bar() {
        cout << "BA::bar() ..." << endl;
    }
};

int main()
{
    BA b;  ——— これはダメ

    return 0;
}
```

次に，このサンプルプログラムをコンパイルしたところを示す．

図23-5▼

```
% g++ go.C
go.C: In function `int main()':
go.C:18: cannot declare variable `b' to be of type `BA'
go.C:18:   since the following virtual functions are abstract:
go.C:18:         int A::foo()
% _
```

このように，抽象クラスは，どこかの派生クラスで純粋仮想関数の本体を定義して実体化しなければならない．

第24章 動的変数

動的変数を使うと，実行時に変数を生成したり，廃棄したりすることができる．この章では，まず，基本型変数を動的に生成・廃棄する方法を説明する．次に，ユーザー定義型の例として，配列を動的に生成・廃棄する．

1 動的変数

変数の生成は，プログラムを書くときに決定される．たとえば，プログラムの中に次のような変数宣言を書くと，プログラム実行時にその変数が作成される．

```
int n;
```

つまり，プログラムでどのような変数を使うかは，プログラムを書くときに決定しなければならない．

それに対して，プログラムの実行時に必要に応じて変数を生成したり，不要になった変数を削除することもできる．それが**動的変数**である．intやcharなどの基本型変数やクラス型の変数（オブジェクト）を実行時に生成したり，廃棄したりすることができる．たとえば，次のようなことが可能である．

```
if (ユーザーの入力が条件を満たす)
    変数を生成する ―――――――――――――――― (1)
```

(1) の部分が実行されるかどうかは，そのときにおけるユーザーの意思で決まる．あるときは実行されるかもしれないし，あるときは実行されないかもしれない．動的変数を使えば，(1) のような場面で変数を生成することができる．

動的変数は，プログラムの実行開始時には存在しない（メモリを消費しない）．そして，実行中に必要があれば，その場で生成することができる（そのとき初めてメモリを消費する）．実行中に必要がなくなれば廃棄することができる（メモリの消費を回復できる）．

——これが，**動的変数**である．たとえば，あるボタンを押すと別のGUI部品が現れるというような処理を行うとき，そのGUI部品に相当するオブジェクトを動的に生成すれば，必要なときだけメモリを消費することができる．

動的変数を使えば，必要なメモリを必要なときだけ確保できるので，無駄なメモリの消費を抑えることができる．

2 基本型への適用

動的変数は，クラス型の変数（オブジェクト）だけでなく，intやcharなどの基本型変数にも適用できる．この節では，基本型の変数を動的に生成・削除する方法を説明する．

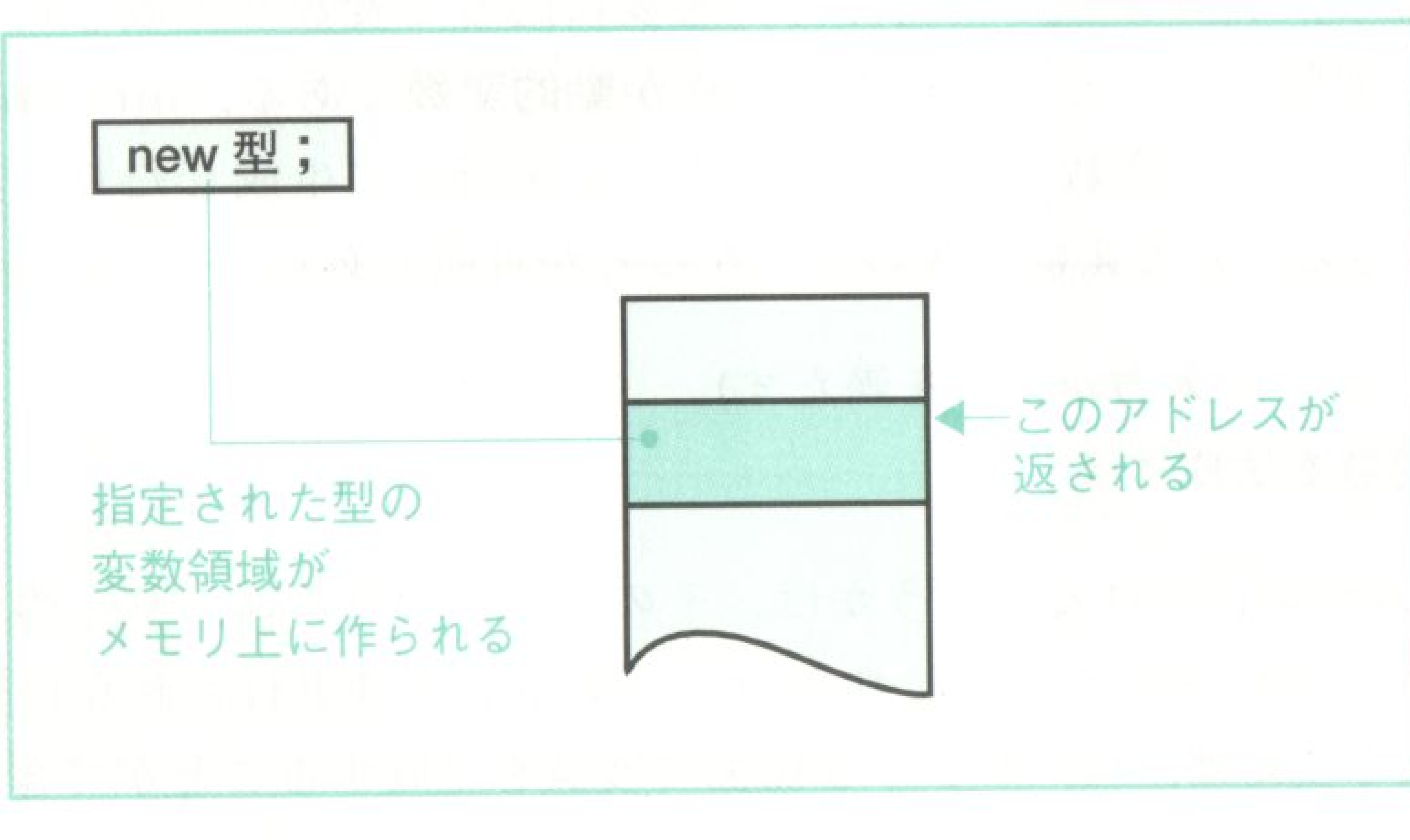
2.1◆動的変数の生成 new

動的変数を生成するにはnewという演算子を使う．

```
new 型
```

この結果，指定した型の変数がメモリ上に確保され，そのアドレスが返される．

図24-1▶



2.2◆動的変数には名前がない

ここで注意したいことは，動的変数には名前がないということである．一般的な変数なら，変数宣言をするときにその名前が決まる．

```
int hello;
```

たとえば，上のような宣言ならhelloが変数の名前なので，この変数名を使って値を代入したり，その値を参照することができる．しかし，動的変数を生成するときは名前を指定しないので，名前がない．

```
new int;
```

指定するのは型だけ
名前は指定しない

たとえば，上の例ならint型の動的変数を生成しているが名前がない．つまり，このままでは，せっかく生成した動的変数を使うことができない．代入することも，値を参照することもできない．

2.3◆ポインタを使う

動的変数には名前がないので，代わりにポインタを使う．

第2.1節で説明したようにnewは作成した動的変数のアドレスを返す．そこで，そのアドレスをポインタで受け取る．そうすれば，そのポインタを使って（逆参照演算子の * を使う）動的変数を操作することができる．

たとえばint型の動的変数を使うなら，int型のポインタを用意する．

```
int *p;  ―――― (1)
```

そして，newの実行結果を，このpに受け取る．

```
p = new int;  ―――― (2)
```

あるいは，この（1）と（2）を，次のように1つにまとめてもよい．

```
int *p = new int;
```

こうすれば，pが生成された動的変数のアドレスを覚えているので，次のようにして値を代入することができる．

```
*p = 100;
```

あるいは，値を読み出すことができる．

```
cout << *p << endl;
```

動的変数の生成をまとめると，次のようになる．

```
Type *p = new Type;
```

ここで，Typeは任意の型である．

注意　メモリが足りなくて指定された型の動的変数を確保できないとき，newは0（ゼロ）を返す．

2.4◆動的変数の削除 delete

動的変数を削除して割り当てられたメモリを解放するにはdeleteを使う．

```
delete ポインタ;
```

deleteの後ろにはポインタを指定する．このポインタが指す動的変数が削除される．このとき面白いのは，削除するメモリの大きさ，あるいは変数の型を指定しなくてもよいということである．newで確保した適切なメモリ量が自動的に計算されて削除される．

new以外で，割り当てられた以外の領域を指定したり，deleteを2回実行したりしても，コンパイラはそのエラーを検出できないので注意．

2.5◆サンプルプログラム

次に動的変数を使ったサンプルプログラムを示す．

【サンプルプログラム】

これは，動的変数を確保し，その変数を使い，削除するまでの一連の流れを示すものである．

```
#include <iostream.h>

int main()
{
    int *p = new int;  ――――――――――― 動的変数の作成
    *p = 1024;  ―――――――――――――― 値の代入
    cout << *p << endl;  ――――――――― 値を読み出す
    delete p;  ――――――――――――――― 動的変数の削除

    return 0;
}
```

【実行例】

```
% a.out
1024
% _
```

3 配列への適用

前の節では，基本型変数を動的に生成・削除する方法を説明した．動的変数は，基本型のみならず，（ユーザー定義型の1つである）**配列**にも適用することができる．すなわち，配列変数を動的に生成・削除することもできる．

これは，強力である．**動的配列**を使えば，実行時にサイズを決めなければならないという配列の欠点を隠すことができるからである．

たとえば，第15章「**配列**」の第4節で，複数のデータの合計を求めるサンプルプログラムを示した．そのプログラムは配列を使ったため，計算できるデータ数の最大数をプログラム時に決めなければならないという欠点を持っていた．この節では，動的配列を使って，その改良版を作成する．

3.1◆動的配列

動的配列を作成するには，newを実行するとき，型名の後ろに大カッコ [] を付けて，作成する配列の要素数を指定する．

```
int *d = new int[10];
```
（↑ ここ）

この宣言によりint型でサイズが10の動的配列が作られる．newで配列を確保すると，最初の要素を指すアドレスが返される．上の宣言では，newが返すアドレスをポインタで受け取っている．

この宣言では，配列のサイズは10になり，10個のint型データを記憶することができる．

```
d[0] ―――― 1番目のデータ
d[1] ―――― 2番目のデータ
……
d[9] ―――― 10番目のデータ
```

動的配列が強力なのは，[] の中に変数を指定することができることである．

```
int size = 10;
int *d = new int[size];
```
（↑ 変数でサイズを指定）

変数が使えるということは，変数の値を変えることにより，いろいろなサイズの配列を宣言できるということである．しかも，変数の値を実行時に決めることができるので，配列のサイズを実行時に決めることができる．

```
cin >> size;             ―――― 実行時にサイズを決める
int *d = new int[size];  ―――― そのサイズの動的配列を確保
```

動的配列を削除するときは，deleteとポインタの間に大カッコ [] を入れる．

```
delete [] d;
```

3.2◆入力データの保存と加算（第2版）

次に示すのは，第15章「配列」の第4節で示した，複数のデータの合計を求めるプログラムの改良版である．計算できるデータの最大数を実行時に決めることができる．

【サンプルプログラム】

```
#include <iostream.h>

int main()
{
    int size, i, ndata, sum = 0;

    cout << "最大データ数? ";  ――――― 入力できるデータの最大数を決める
    cin >> size;
    int *d = new int[size];

    cout << "複数データの合計を求めます" << endl
         << "入力の最後は負数を入力してください" << endl;

    for (i = 0; i < size; i++) {
        cout << "データ" << i + 1 << " ? ";
        cin >> d[i];
        if (d[i] < 0)
            break;
    }
    ndata = i;

    cout << endl;
    for (i = 0; i < ndata; i++) {
        cout << "データ" << i + 1 << " = " << d[i] << endl;
        sum += d[i];
    }
    cout << endl
         << "データ数 = " << ndata << endl
         << "合計 = " << sum << endl;

    delete [] d;

    return 0;
}
```

【実行例】

```
% a.out
最大データ数? 5 ――――――――――――― この場合、最大5件まで計算できる
複数データの合計を求めます
入力の最後は負数を入力してください
データ1 ? 1 ――――――――――――――― 途中で止める場合は負数を入力する
```

```
データ2 ? 2
データ3 ? 3
データ4 ? 4
データ5 ? 5

データ1 = 1 ──── 検算用に入力したデータが表示される
データ2 = 2
データ3 = 3
データ4 = 4
データ5 = 5

データ数 = 5
合計 = 15
% _
```

第25章 動的オブジェクト

この章では，動的変数をオブジェクトに適用する．すなわち，オブジェクトをプログラムの実行時に生成・廃棄する方法を説明する．また，動的オブジェクトについて問題になる点を説明する．

1 動的オブジェクト

実行時に動的に作られたオブジェクトを**動的オブジェクト**という．この節では，動的オブジェクトを生成・廃棄する手順を説明する．

1.1◆作成と削除

オブジェクトを動的に作成するときも，一般的な変数と同じようにnew演算子を使う．ただ，型名のところにクラス型を指定するだけである．

```
Classname *pointer = new Classname;
```

また，動的オブジェクトを削除するにはdelete演算子を使う．

```
delete pointer;
```

次に動的オブジェクトの作成から削除までの流れを示す．

```
Classname *pointer;           ―― クラス型のポインタを用意
pointer = new Classname;      ―― 動的オブジェクトを作成
                                 そのアドレスをポインタで受け取る
        ◀――――――――――――――――――――― ここで動的オブジェクトを使う
delete pointer;               ―― 不要になった動的オブジェクトを
                                 削除する
```

1つ注意しておくと，動的オブジェクトはポインタを通して操作される．したがって，動的オブジェクトのメンバーを参照するときは，ピリオドではなく->を使う．

```
pointer->func();
```

1.2◆サンプルプログラム

動的オブジェクトをテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

class Foo {                    ―― この型の動的オブジェクトを作る
public:
    void show() {
        cout << "Foo::show running ..." << endl;
    }
};

int main()
{
    Foo *pfoo = new Foo;       ―― 動的オブジェクトを作成
    pfoo->show();              ―― メンバー関数の参照は->を使う
    delete pfoo;               ―― 動的オブジェクトを削除

    return 0;
}
```

【実行例】

```
% a.out
Foo::show running ...
% _
```

2 コンストラクタとデストラクタ

動的オブジェクトも普通のオブジェクトと同じように，作成されるときにコンストラクタが自動的に実行される．たとえば，Fooというクラスがあって，コンストラクタが定義されているとき，

Foo *pfoo = new Foo;

のようにして動的オブジェクトを作成すると，コンストラクタが実行される．

またdelete演算子を使って動的オブジェクトを削除すると，自動的にデストラクタが実行される．たとえば，次のようにするとデストラクタが実行される．

delete pfoo;

以上をテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

class Foo {
public:
    Foo() { ──────────────── コンストラクタ
        cout << "constructor Foo()" << endl;
    }

    ~Foo() { ──────────────── デストラクタ
        cout << "desstructor Foo()" << endl;
    }
};

int main()
```

```
{
    Foo *pfoo = new Foo;      ―― 動的オブジェクトを作成
    delete pfoo;              ―― 動的オブジェクトを削除

    return 0;
}
```

【実行例】

```
% a.out
constructor Foo()      ―― コンストラクタが実行されている
desstructor Foo()      ―― デストラクタが実行されている
% _
```

3 引数付きのコンストラクタ

コンストラクタが引数をとるときは，newの最後に（）を付けて引数を指定する．

```
Foo *pfoo = new Foo(8001);
```

（8001：コンストラクタに渡す引数）

次に例を示す．

【サンプルプログラム】

```
#include <iostream.h>

class Foo {
public:
    Foo(int n);      ―― 引数付きのコンストラクタ
};

Foo::Foo(int n)
{
    cout << "constructor Foo() running ..." << endl
         << "argument = " << n << endl;
}
```

```
int main()
{
    Foo *pfoo = new Foo(8001);  ―――― コンストラクタに引数を渡す
    delete pfoo;

    return 0;
}
```

【実行例】

```
% a.out
constructor Foo() running ...
argument = 8001
% _
```

4 コンポジション

コンポジションというのは，他のクラスをメンバーに持つクラスのことである．他のクラスをメンバーに加えることにより，複数のクラスの機能を持つ高機能なクラスを作ることができる．

この節では，コンポジションを作るときの注意点を説明する．

4.1◆コンポジション

他のクラスをメンバーに持つクラスをコンポジションという．たとえば，次のクラスBは，クラスAをメンバーに持っているのでコンポジションである．

```
class A {
};

class B {
    A obj_a;  ―――― クラス型のメンバー
};                 obj_aはメンバーオブジェクト
```

以下，obj_aを**メンバーオブジェクト**と呼ぶことにする．メンバーオブジェクトは，クラスのメンバーであるオブジェクトのことである．

4.2◆is a関係とhas a関係

コンポジションは，内部に別のオブジェクトを持つので，2つのオブジェクトの機能を使うことができる．そういう意味で，コンポジションは，第22章で説明した継承に似ている．

継承は，親クラスの機能を使うことができる．コンポジションも，メンバーにしたクラスの機能を使うことができる．他のクラスの機能を使うことができるという点で，コンポジションも継承も似ている．

そこで問題になるのが，あるクラスの機能を使って新しいクラスを作るとき，コンポジションを使ったほうがよいのか，継承を使ったほうがよいのかという問題である．いわゆる**コンポジションと継承の使い分け**である．

――この使い分けは，微妙である．通常は，オブジェクト間の関係を考えて使い分けを行う．

●has a関係

コンポジションの場合は，コンポジションが他のオブジェクトを所有する関係にある．この関係を**has a関係**という．コード的には，他のオブジェクトの機能（要するに他のクラスのメンバー関数）を多用する場合は，コンポジションが向いている．

●is a関係

一方，継承は，他のオブジェクトの機能の拡張である．継承されたオブジェクトは，継承元のすべての機能を含む．つまり，両者は祖先を同じくする同じ仲間で，一方が一方の一種であるという関係になっている．この関係を**is a関係**という．

継承は，第23章以下で説明した多態性を使い，同じ一族をまとめて処理する場合に向いている．

4.3◆コンストラクタ

コンポジションは，メンバーにオブジェクトを持っているので，コンポジションが作られるとき，2つのオブジェクトが作られることになる．そのとき，メンバーオブジェクトが先に作られるのか，自分自身が先に作られるのかという問題がある．

```
class B {
    A obj_a; ―――――― メンバーオブジェクト
};

B b; ―――――――――― このときbが先に作られるか？
                     obj_aが先に作られるか？
```

このことはコンストラクタの実行とも関わり合いを持つ．すなわち，メンバーオブジェクトのコンストラクタが先に実行されるのか，それとも自分自身のコンストラクタが先に実行されるのかという問題である．

以上の点を検証するためのサンプルプログラムを次に示す．

【サンプルプログラム】

これは，コンポジションにおいて自分自身とメンバーのどちらのコンストラクタが先に実行されるかを検証するためのプログラムである．

クラスAはコンストラクタだけを持つ．またクラスBは，クラスAのメンバーとコンストラクタを持つコンポジションである．どちらのコンストラクタも，実行されるとクラス名を表示する．これにより，どのコンストラクタが実行されたかがわかるようになっている．

```
#include <iostream.h>

// このクラスをメンバーオブジェクトにする
class A {
public:
    A() {
        cout << "A::A() running..." << endl;
    }
};

// コンポジション
class B {
    A obj_a;
public:
    B() {
        cout << "B::B() running..." << endl;
    }
};
```

```
int main()
{
    B b;

    return 0;
}
```

【実行例】

```
% a.out
A::A() running... ―――― 先にメンバーオブジェクトが作られる
B::B() running... ―――― 次にコンポジションが作られる
% _
```

この実行結果を見ると，まずメンバーオブジェクトのコンストラクタが実行され，続いてコンポジション自身のコンストラクタが実行されているのがわかる．このようにコンポジションにおいては，まずメンバーが先に作られる．そして，その後でコンポジションのオブジェクトが作られる．

4.4◆デストラクタ

前項では，コンポジションにおけるコンストラクタを検証した．ここでは，コンポジションにおけるデストラクタについて調べる．

【サンプルプログラム】

これは，コンポジションにおいて自分自身とメンバーのどちらのデストラクタが先に実行されるかを検証するためのプログラムである．

```
#include <iostream.h>

// このクラスをメンバーオブジェクトにする
class A {
public:
    ~A() {
        cout << "A::~A() running..." << endl;
    }
};

// コンポジション
```

```
class B {
    A obj_a;
public:
    ~B() {
        cout << "B::~B() running..." << endl;
    }
};

int main()
{
    B b;

    return 0;
}
```

【実行例】

```
% a.out
B::~B() running... ———————— 先にコンポジションが廃棄される
A::~A() running... ———————— 次にメンバーオブジェクトが廃棄される
% _
```

この実行結果を見ると，最初にコンポジションのデストラクタが実行され，次にメンバーオブジェクトのデストラクタが実行されているのがわかる．このように，コンポジションにおいては，まずコンポジション自身が削除され，次にメンバーオブジェクトが削除される．

4.5◆引数付きのコンストラクタ

メンバーオブジェクトのコンストラクタが引数をとるとき，コンポジションからその引数をどのように渡せばよいかという問題がある．

ちょっと考えると，コンポジションのコンストラクタの中で引数を渡せばよいと思われるが，これはまずい．なぜなら第4.3節で検証したように，コンポジションのコンストラクタが実行されるときには，すでにメンバーオブジェクトのコンストラクタは実行済みだからである．

この問題を回避するため，**メンバーイニシャライザ**という仕掛けが用意されている．これは，コンポジションのコンストラクタで，本体のコードが始まる前にメンバーオブジェクトのコンストラクタを呼び出すためのものである．

メンバーイニシャライザ

```
B(int n) : obj_a(n) {
    cout << "B::B() running..." << endl;
}
```

この書き方は，第22章「継承」の第6.2節で扱ったイニシャライザに似ている．

コンポジションが複数のメンバーオブジェクトを持つときには，メンバーイニシャライザで，それらのコンストラクタをカンマで区切って並べればよい．

【サンプルプログラム】

これは，メンバーイニシャライザの書き方を示すサンプルプログラムである．

```
#include <iostream.h>

class A {
    int data;
public:
    A(int n) {            ―――― 引数をとる
        cout << "A::A() running..." << endl;
        data = n;
    }

    int get() {
        return data;
    }
};

class B {
    A obj_a;
public:
    B(int n) : obj_a(n) {     ―――― メンバーイニシャライザ
        cout << "B::B() running..." << endl;
    }

    void show() {
      cout << obj_a.get() << endl;
    }
};

int main()
{
```

```
    B b(1000);  ──── 1000がメンバーオブジェクトに渡される
    b.show();

    return 0;
}
```

【実行例】

```
% a.out
A::A() running...
B::B() running...
1000 ──── 正しく初期化されているのがわかる
% _
```

第26章 演算子の多重定義

基本型のデータは，演算子を使っていろいろな演算ができる．一方，ユーザー定義型は演算子を使うことができない．**演算子の多重定義**という機能を使うと，クラスで演算子を定義して使うことができる．

1 クラスの代入

一般に，クラスでは演算が行えない．しかし，1つだけ例外があって，演算子の多重定義を使わないでも**代入演算**の＝だけは使用可能である．

```
Vect v1, v2;  ―――― Vectは，あるクラス
v1 = v2;      ―――― 代入が可能
```

クラスの代入では，対応する各メンバー変数がそれぞれ代入される．たとえば，いま示したVectは，2つのメンバー変数x，yを持っているとする．

```
class Vect {
    int x, y;  ―――― 2つのメンバー変数
};
```

クラスVectは，それらのメンバー変数に値を設定するため，setというメンバー関数を持っている．

```
class Vect {
    ……
```

```
public:
    void set(int a = 0, int b = 0) { ―――メンバー変数を初期化する
        x = a;
        y = b;
    }
};
```

また，メンバー変数の値を表示するshowというメンバー関数を持っている．

```
class Vect {
    ……
public:
    ……
    void show() {
        cout << "(x, y) = (" << x << ", " << y << ")" << endl;
    }
};
```

そのクラスVectのオブジェクトを2つ作る．

```
Vect v1, v2;
```

そして，オブジェクトv2の方だけを初期化する．

```
v2.set(100, 200); ―――v2のメンバー変数に初期値を与える
```

オブジェクトv1のメンバー変数は初期化されていない．しかし，次のようにv1にv2を代入すると，v1のメンバー変数はv2と同じ値になる．

```
v1 = v2; ―――代入が可能
```

以上をテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

class Vect {
    int x, y;
public:
    void set(int a = 0, int b = 0) {
        x = a;
```

```
        y = b;
    }

    void show() {
        cout << "(x, y) = (" << x << ", " << y << ")" << endl;
    }
};

int main()
{
    Vect v1, v2;
    v2.set(100, 200);
    v1 = v2;
    v1.show();

    return 0;
}

% a.out
(x, y) = (100, 200)    ―― v1のメンバー変数
% _
```

v1のメンバー変数の値が，v2のそれと同じになっているのがわかる．

2 2項演算子の多重定義（メンバー関数版）

一般にオブジェクトは，代入演算子=以外の演算子を使うことはできない．しかし，クラス宣言の中で演算子の使い方を定義すると，オブジェクトでその演算子が使えるようになる．3項演算子を除く既存の演算子なら，どれでも多重定義できる．

演算子を多重定義するには，次のいずれかの関数を使う．

メンバー関数
フレンド関数

手始めに，2項演算子の多重定義から説明を始める．

2.1◆演算子メンバー関数の書式

2項演算子とは，演算の対象となるオペランドが2つある演算子である．たとえば，次の + が2項演算子である．

```
1000 + 2000
```

ここでは，Vectクラスに2項演算子の + を多重定義する例を説明する．メンバー関数を使って演算子を多重定義するには，次のようなメンバー関数をクラスに追加する．

メンバー関数の型	：そのクラスの型 この例ではVect
メンバー関数の名前	：演算子の前にoperatorを付けたもの この例ではoperator+
引数	：そのクラスの引数を1つとる この例ではVect v 引数の名前は何でもよい

したがって，次のようなメンバー関数を定義すればよい．

```
class Vect {
    ……
public:
    Vect operator+(Vect v);    ←――― ここ
    ……
};
```

2.2◆メンバー関数の本体

メンバー関数の本体には，その演算子で処理される内容を記述する．これは書き方が決まっていて，次のような流れで処理を書く．

① そのクラス自身が左オペランドになる
すなわち，そのクラスのメンバー変数が左オペランドのメンバー変数になる
② 演算子メンバー関数の引数が右オペランドになる
したがって，もし右オペランドの値を書き換えるような処理を行うなら，

引数を参照引数にするとよい
③ ①のメンバー変数と②のメンバー変数の間で演算を行う
④メンバー関数の中で，そのクラスのオブジェクトを適当な名前で作る
⑤ ③の演算結果を④のオブジェクトに代入する
⑥ ④のオブジェクトを関数値にする

まず①についてであるが，Vectクラスでは，メンバー変数のx，yが左オペランドのメンバー変数になる．

```
class Vect {
      int x, y;          // これが左オペランド
public:
      .........
};
```

また②については，引数のvが右オペランドになる．

```
                         右オペランド
                              ↓
Vect Vect::operator+(Vect v)
{
      ............
}
```

クラスVectの+演算子では，それぞれのオブジェクトの対応するメンバーどうし（xとyがある）を加算することにする．したがって，③には次のようなコードを書けばよい．

```
x + v.x        // xはxどうしで加算する
y + v.y        // yはyどうしで加算する
```

④は，たとえば次のようなtempを宣言する．

```
Vect Vect::operator+(Vect v)
{
      Vect temp;   ←―――― ここ
      .........
}
```

このtempに③の演算結果を受け取る．それが⑤である．

```
Vect Vect::operator+(Vect v)
{
    Vect temp;

    temp.x = x + v.x;   ← ここと
    temp.y = y + v.y;   ← ここ
}
```

そして，⑥によりtempをreturnすれば関数の出来上がりである．

```
Vect Vect::operator+(Vect v)
{
    Vect temp;

    temp.x = x + v.x;
    temp.y = y + v.y;

    return temp;   ← ここ
}
```

2.3◆2項演算子 +

以上をテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

これは，クラスVectに2項演算子+を組み込む例である．この演算子は，対応するメンバー変数どうしを加算する．

```
#include <iostream.h>

class Vect {
    int x, y;
public:
    Vect(int a = 0, int b = 0) {
        x = a;
        y = b;
```

```
    }

    void show() {
        cout << "(x, y) = (" << x << ", " << y << ")" << endl;
    }

    Vect operator+(Vect v); ───────── 演算子メンバー関数
};

Vect Vect::operator+(Vect v) ───────── 演算子メンバー関数の本体
{
    Vect temp;

    temp.x = x + v.x;          // 一時変数=自身+引数
    temp.y = y + v.y;          // の演算を行い

    return temp;               // 一時変数を返す
}

int main()
{
    Vect v, ───────────────────── 結果を受け取る
        v1(10, 20), v2(100, 200); ───── 2つのオペランド

    v = v1 + v2; ───────────────── 演算子+を使う
    v.show();

    return 0;
}
```

【実行例】

```
% a.out
(x, y) = (110, 220)
% _
```

この結果を見ると，メンバー変数どうしが加算されているのがわかる．

3 フレンド関数

フレンド関数を使って演算子を多重定義することもできる．**フレンド関数**というのは，クラスのメンバー関数ではない．クラスの外で宣言される一般的な関数である．にもかかわらず，クラスのメンバー関数と同じようにクラスの非公開メンバー（privateメンバーとprotectedメンバー）を参照することができる．

メンバー関数があれば，フレンド関数はいらないように見える．メンバー関数がありながら，それと同じように機能するフレンド関数が存在するのはなぜだろう？

この節では，フレンド関数の使い方と，その存在意義を扱う．

3.1◆フレンド関数の宣言

最初に，フレンド関数の作り方から始める．フレンド関数は，一般的な関数である．したがって，普通の関数と同じようにクラスの外で宣言する．たとえば，次はshowという名前のフレンド関数である．

```
void show()
{
      ............
}
```

フレンド関数の引数には，特に制限がない．しかし，クラスのprivateメンバーを参照するためには，そのクラスのオブジェクトを引数にとるようにする．たとえばフレンド関数showをクラスAのフレンド関数にするなら次のようにする．

```
void show(A &obj_a)——— A型の引数をとる
{
      ............
}
```

この場合，フレンド関数showは引数に指定されたobj_aのprivateメンバーを参照することができる．

ところで，ここまでの宣言ではshowは一般の関数と何ら変わりがなく，このままではフレンド関数にはならない．showをフレンド関数にするためには，フレンドにしたいクラスの中でfriendを付けて関数宣言すればよい．たとえば，showを

クラスAのフレンド関数にするには次のようにする．

```
class A {
    int data;
public:
    friend void show(A obj_a);
};
```

int data; ── フレンド関数は，このメンバー変数を参照できる

friend void show(A obj_a); ── showをフレンド関数にする

宣言の位置は，どのセクションで行ってもよい．なお，フレンド関数の本体の宣言は，メンバー関数ではないのでA::を付ける必要はない．

これは不要（A::）

```
void A::show(A obj_a)
{
    ............
}
```

フレンド関数は，引数で受け取ったオブジェクトの非公開メンバーを参照できる．たとえば次は，obj_aのprivateメンバーdataの値を出力している．

```
void show(A obj_a)
{
    cout << obj_a.data << endl;
}
```

以上をテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

class A {
    int data;
public:
    A(int n = 0) {
        data = n;
    }
```

```
    friend void show(A obj_a);  ——— showがフレンドであることを宣言
};

void show(A obj_a)  ——— フレンド関数本体の定義
{
    cout << obj_a.data << endl;
}

int main()
{
    A a(1000);  ——— クラスAのオブジェクトを作り
    show(a);  ——— フレンド関数に渡す

    return 0;
}
```

【実行例】

```
% a.out
1000  ——— この値はprivateメンバーdataの値
% -
```

3.2◆フレンド関数をインラインにする

一般の関数をフレンド関数にするには，クラスの中でfriendを付けて関数宣言するわけであるが，そのときついでに関数の本体を定義することもできる．すると，そのフレンド関数をインライン関数にすることができる．たとえば，いまのshowをインライン関数にするには，次のようにする．

```
class A {
    int data;
public:
    friend void show(A obj_a) {  ——— 関数本体も定義する
        cout << obj_a.data << endl;
    }
    …………
};
```

インライン関数については，すでに何回か出てきたが，関数の呼び出し位置にコードが埋め込まれるので，実行速度を速くすることができる．

次に第3.1節のshowをインラインコードに書き換えたものを示す．

【サンプルプログラム】

```
#include <iostream.h>

class A {
    int data;
public:
    A(int n = 0) {
        data = n;
    }

    friend void show(A obj_a) {
        cout << obj_a.data << endl;
    }
};

int main()
{
    A a(1000);
    show(a);

    return 0;
}
```

【実行例】

```
% a.out
1000 ———————————— この値はprivateメンバーdataの値
% -
```

3.3◆オブジェクト間でデータを共有

メンバー関数の代わりにフレンド関数を使う場面としては，同じクラスの複数のオブジェクトの間でデータを共有するというのがある．フレンド関数を使えば，複数のオブジェクトの非公開部を同時に参照することができるからである．

たとえば，次に示すcopyというフレンド関数は，クラスAのオブジェクトどうしでコピーを行う．

```
friend void copy(A &a1, A a2) {
    a1.data = a2.data;
}
```

copy(a, b); のように使用すると，bのメンバー変数がaのメンバー変数にコピーされる．

なお，copyの宣言で引数のa1を参照引数にしていることに注意．

```
                    ここ
                     ↓
friend void copy(A &a1, A a2)
```

これは，a2をa1にコピーするのでa1の値を書き換えるからである．参照引数にしないと，その書き換えを関数の呼び出し側に反映させることができない．a2の方は書き換えないので，参照引数にする必要はない．

以上をテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

class A {
    int data;
public:
    A(int n = 0) {
        data = n;
    }

    friend void show(A &obj_a) {
        cout << obj_a.data << endl;
    }

    friend void copy(A &a1, A a2) {  ――― 追加したフレンド関数
        a1.data = a2.data;  ――――――― 2つのオブジェクトを参照
    }
};
```

```
int main()
{
    A a, b(2000);

    copy(a, b);  ―― bをaにコピー
    show(a);

    return 0;
}
```

【実行例】

```
% a.out
2000  ―― コピーされている
% _
```

3.4◆複数のクラスで共通のフレンド関数を持つ

フレンド関数の存在意義を示すもう1つの例として，複数のクラスで共通のフレンド関数を持つというのがある．そのフレンド関数は，複数のクラスの非公開部を参照することができる．このレベルになると，もはや普通のメンバー関数では不可能である．

たとえば，次のようにA，Bという2つのクラスがあって，それぞれprivateメンバーを持っているとする．

```
class A {
    int data_a;  ―― クラスAのprivateメンバー
public:
    ............
};

class B {
    int data_b;  ―― クラスBのprivateメンバー
public:
    ............
};
```

1つのメンバー関数でdata_a（クラスAのprivateメンバー）とdata_b（クラスBのprivateメンバー）を同時に参照するのは不可能である．しかし，フレンド関数

なら，それが可能である．たとえば，次のフレンド関数showのように．

```
void show(A a, B b)
{
        cout << "A::data_a = " << a.data_a << endl
             << "B::data_b = " << b.data_b << endl;
}
```

showは，A，B両クラスのフレンド関数なので，それぞれのクラスのオブジェクトを引数にとる．showは，それぞれのクラスでfriend宣言する必要がある．

```
class A {
     …………
     friend void show(A a, B b);  ◀――― ここと
};

class B {
     …………
     friend void show(A a, B b);  ◀――― ここ
};
```

1つ重要な注意がある．showをclass Aでfriend宣言するとき，Bを引数にしているが，この段階では，まだclass Bの宣言が行われていない．そのため，このままではエラーになってしまう．

そこで，class Aの宣言の前にclass Bの仮宣言（本体のない宣言）を行っておく．

```
class B; ――――――――――――――――― クラスBの仮宣言

class A {
     …………
     friend void show(A a, B b); ――― ここでBを使っている
};

class B { ――――――――――――――――― クラスBの本宣言
     …………
     friend void show(A a, B b);
};
```

以上をテストするためのサンプルプログラムを次に示す.

【サンプルプログラム】

```
#include <iostream.h>

class B;

class A {
    int data_a;
public:
    A(int n = 0) {
        data_a = n;
    }

    friend void show(A a, B b); ——— 共通フレンド関数
};

class B {
    int data_b;
public:
    B(int n = 0) {
        data_b = n;
    }

    friend void show(A a, B b); ——— 共通フレンド関数
};

void show(A a, B b)
{
     cout << "A::data_a = " << a.data_a << endl
          << "B::data_b = " << b.data_b << endl;
}

int main()
{
    A a(1000);
    B b(2000);

    show(a, b); ——————————— 1つの関数で2つのクラスを参照する

    return 0;
}
```

【実行例】

```
% a.out
A::data_a = 1000
B::data_b = 2000
% _
```

4 2項演算子の多重定義（フレンド関数版）

第2節でメンバー関数を使った2項演算子の多重定義を説明した．この方法では，そのクラス自身が左オペランドになる．このため，左オペランドの型は，必ずそのクラスの型でなければならないという制限がある．

x op y;
↑
演算子

たとえば上の例でopがクラスVectで定義された演算子とすると，xの型はVectでなければならない．

フレンド関数を使って演算子を定義すると，この制限が解除される．すなわち，左オペランドの型を自由に定義することができる．

4.1◆フレンド関数による演算子の定義

Vectクラスに新しい2項演算子を定義する．新しい + は，左オペランドがintである．

300 + vectobj
↑
int型

この演算により，右オペランドのVectオブジェクトの各メンバー変数の値が左オペランドの値だけ大きくなる．

このような定義は，メンバー関数ではできない．フレンド関数を使う．次が，

そのフレンド関数である.

```
Vect operator+(int n, Vect v)
{
    ............
}
```

フレンド関数版では，引数が2つ必要である（第2節のメンバー関数版は引数が1つであった）．第1引数が左オペランド，第2引数が右オペランドになる．この例では，左オペランドがintになるので，第1引数の型はintである．

```
Vect operator+(int n, Vect v)
```

左オペランド
型はint

2つある引数のうち，いずれかの引数はフレンド関数を定義しているクラスと同じ型でなければならない．さもないと，クラスの演算子を多重定義する意味がない．

```
Vect operator+(int n, Vect v)
```

右オペランド
型は，その演算子が属するクラスと同じ

フレンド関数の返す値が，その演算子の返す値になる．この例では，結果がVect型なのでVectである．

```
Vect operator+(int n, Vect v)
```

この演算子が返す値の型

フレンド関数本体の定義は次のようになる．適当なVect型の変数を作り，そこに演算結果を代入して返すようにする．

```
Vect operator+(int n, Vect v)
{
    Vect temp;

    temp.x = n + v.x;        // 一時変数 = 引数1 + 引数2
```

```
        temp.y = n + v.y;          // の演算を行い

        return temp;               // 一時変数を返す
    }
```

最後に，この関数がフレンド関数になるようにclass Vectの中でfriend宣言する．

```
    class Vect {
        ............
        friend Vect operator+(int n, Vect );
    };
```

これで完成である．

4.2◆サンプルプログラム

第4.1節で定義した+演算子をテストするためのサンプルプログラムを次に示す．

【サンプルプログラム】

```
#include <iostream.h>

class Vect {
    int x, y;
public:
    Vect(int a = 0, int b = 0) {
        x = a;
        y = b;
    }

    void show() {
        cout << "(x, y) = (" << x << ", " << y << ")" << endl;
    }

    friend Vect operator+(int n, Vect v);
};

Vect operator+(int n, Vect v)
{
    Vect temp;

    temp.x = n + v.x;          // 一時変数 = 引数1 + 引数2
```

```
    temp.y = n + v.y;           // の演算を行い

    return temp;                // 一時変数を返す
}

int main()
{
    Vect v, v0(100, 200);

    v = 3 + v0;
    v.show();

    return 0;
}
```

【実行結果】

```
% a.out
(x, y) = (103, 203)
% _
```

各メンバー変数の値が，それぞれ+3されているのがわかる．

5 単項演算子の多重定義(メンバー関数版)

続いて，単項演算子（対象となるオペランドが1つ）の多重定義に進む．単項演算子についても，メンバー関数とフレンド関数の2通りのやり方がある．

ここでは，メンバー関数を使ってVectクラスに単項演算子の-を組み込む方法を説明する．この単項演算子は，Vectオブジェクトの各メンバー変数の符号を逆転するようにインプリメントする．

Vect v(100, 200);

―― Vect型のオブジェクトを作る
メンバー変数のx, yを100, 200で初期化

v = -v; ―― vの各メンバー変数の符号を逆転する

メンバー関数で2項演算子を定義するときは引数を1つとったが，単項演算子の場合は引数をとらない．

```
Vect Vect::operator-()
{
}
```

関数の本体では，Vect型のローカル変数を宣言し，そこに演算結果を作る．そして，それを関数としてreturnするようにする．

```
Vect Vect::operator-()
{
    Vect temp;

    temp.x = -x;           // 一時変数 = 自身 + 引数
    temp.y = -y;           // の演算を行い

    return temp;           // 一時変数を返す
}
```

次に完成版のサンプルプログラムを示す．

【サンプルプログラム】

```
#include <iostream.h>

class Vect {
    int x, y;
public:
    Vect(int a = 0, int b = 0) {
        x = a;
        y = b;
    }

    void show() {
        cout << "(x, y) = (" << x << ", " << y << ")" << endl;
    }

    Vect operator-();
};
```

```
Vect Vect::operator-()
{
    Vect temp;

    temp.x = -x;                    // 一時変数 = 自身 + 引数
    temp.y = -y;                    // の演算を行い

   return temp;                     // 一時変数を返す
}

int main()
{
    Vect v(100, 200);

    v.show(); ─────────────────── 初期値を表示
    v = -v; ──────────────────── 単項演算を行う
    v.show(); ─────────────────── 演算後の結果を表示する

    return 0;
}
```

【実行例】

```
% a.out
(x, y) = (100, 200)
(x, y) = (-100, -200) ─────────── 符号が逆転されている
% _
```

6 単項演算子の多重定義(フレンド関数版)

フレンド関数を使って単項演算子を定義することもできる．しかし，2項演算子のときとは違って，メンバー関数版との違いは，ほとんどない．ただ，やり方が変わるだけである．

第5節と同じ単項演算子-をフレンド関数で定義した例を次に示す．

【サンプルプログラム】

```
#include <iostream.h>

class Vect {
```

```
    int x, y;
public:
    Vect(int a = 0, int b = 0) {
        x = a;
        y = b;
    }

    void show() {
        cout << "(x, y) = (" << x << ", " << y << ")" << endl;
    }

    friend Vect operator-(Vect v);
};

Vect operator-(Vect v)
{
    Vect temp;

    temp.x = -v.x;                  // 一時変数 = 自身 + 引数
    temp.y = -v.y;                  // の演算を行い

    return temp;                    // 一時変数を返す
}

int main()
{
    Vect v(100, 200);

    v.show();
    v = -v;
    v.show();

    return 0;
}
```

【実行例】

```
% a.out
(x, y) = (100, 200)
(x, y) = (-100, -200) ―――――――― 符号が逆転されている
% _
```

7 演算子の多重定義

演算子は関数（メンバー関数またはフレンド関数）を使って定義される．引数が異なれば，同じ名前の関数を多重定義することができる．同様に，演算子も引数が異なれば，多重定義することができる．

演算子で「引数が異なる」ということは，「オペランドの型が異なる」ということである．たとえば，Vectクラスに+演算子を定義するにしても，次のようないろいろなバリエーションを多重定義することができる．

```
class Vect {
    ............
    Vect operator+(Vect v);                   ―― (1)
    friend Vect operator+(int n, Vect v);     ―― (2)
    friend Vect operator+(Vect v, int n);     ―― (3)
};
```

(1)，(2)，(3) ともに2項演算子の + を定義している．しかし，いずれも使い方が異なる．

```
Vect v1, v2;  ―― v1, v2はVect型

v1 + v2;
        ―― 第1, 2オペランドともVectなので(1)が適用される

100 + v1;
        ―― 第1オペランドがintなので(2)が適用される

v1 + 200;
        ―― 第2オペランドがintなので(3)が適用される
```

8 まとめ

クラスにおける演算子の多重定義を見てきた．この機能は，クラスを基本型と同じように自由に演算子を定義して動作させるために存在する．ただし，何でもかんでも定義できるわけではなく，それなりに制限がある．

演算子の多重定義における制限

新しい演算子を作ることはできない
　　存在しない演算子を勝手に定義することはできない
演算子のオペランド数を変更することはできない
　　たとえば，単項演算子を2項演算子にすることはできない
演算子の優先順位を変えることはできない
演算子の結合規則を変えることはできない
基本型の演算子の動作を変えることはできない
次の演算子は多重定義できない

演算子	説明
.	クラスのメンバーを指す演算子
::	スコープ解決演算子
? と :	3項演算子

応用プログラミング

PART 3

第27章 分割コンパイル

これまでは，1つのプログラムを1つのファイルで作成してきた．しかし，プログラムを複数のファイルに分けて開発することもできる．ファイルを分割することにより，次のようなメリットが得られる．

- ファイルを分割することで個々のファイルが小さくなるので，プログラムの見通しがよくなる
- 関連する関数やクラスを1つのファイルまとめることにより，プログラムを再利用しやすくなる
- プログラムの修正などで再コンパイルするとき，必要なファイルだけを再コンパイルすればよいので，開発時間を短縮できる

この章では，分割コンパイルを行うときのやり方と注意点を説明する．

1 ファイルの分割

第16章第3節でswapという関数を開発した．この関数は，2つの整数の値を交換するものであった．

```
// xとyの値を交換する（第3版）
void swap(int &x, int &y)
{
    x = y - x;
```

```
        y -= x;
        x += y;
    }
```

このような汎用的な関数は，いろいろなプログラムで利用することができる．しかし，この関数を利用するたびにswapのコードをコピーしてくるのは，あまり生産的ではない．

そこで“util.C”という汎用的な関数を集めたファイルを作成し，そこにswapのコードを書いておく．そして，swapを使うプログラムでは，util.Cをいっしょにコンパイルするようにする．

util.C ―――― swapなど汎用的な関数を集めたファイル
このファイルは書き換えない

main.C ―――― そのアプリケーション専用のコードを収めたファイル
このファイルはアプリケーションごとに書き換える

こうすれば，毎回swapをコピーし直す必要がなくなる．この例ではファイルを2つに分けたが，必要に応じて3つ以上に分けてもかまわない．

2 分割コンパイル

この節では，ファイルを分割して開発するとき，どのようにコンパイルすればよいのかを説明する．

2.1◆オブジェクトファイル

第1章第3節でコンパイルのしかたを説明した．そこでは，ソースファイルをコンパイルすることで実行ファイルが出来上がると説明した．

ソースファイル（拡張子は“.C”）

↓ コンパイラでコンパイル

実行ファイル（デフォルトではa.out）

しかし，実際は実行ファイルが出来上がる前に**オブジェクトファイル**と呼ばれる半完成品の中間ファイルが作られる．オブジェクトファイルのファイル名は，ソースファイルと同じで，拡張子は“.o”になる．たとえばmain.Cというソースファイルをコンパイルすると，main.oというオブジェクトファイルが作れる．

そのオブジェクトファイルを**リンカ**と呼ばれるプログラムで**リンク**することにより，実行ファイルが作られる．

ソースファイル（拡張子は“.C”）

↓ コンパイラでコンパイル

オブジェクトファイル（拡張子は“.o”）

↓ リンカでリンク

実行ファイル（デフォルトではa.out）

g++は，コンパイルとリンクを自動的に実行する．また，実行ファイルがうまく作られると，オブジェクトファイルを自動的に削除する．このため，ソースファイルから，いきなり実行ファイルが作られるように見える．

2.2◆リンカの威力

ファイルを分割してプログラムを開発する際は，それぞれのソースファイルをコンパイルするとき，いきなり実行ファイルを作成するのではなく，途中のオブジェクトファイルまでの作成で止めておく．たとえば，1つのプログラムを，

main.C
util.C

の2つのファイルに分割して開発するとき，main.Cからmain.oを作り，util.Cからutil.oを作成する．

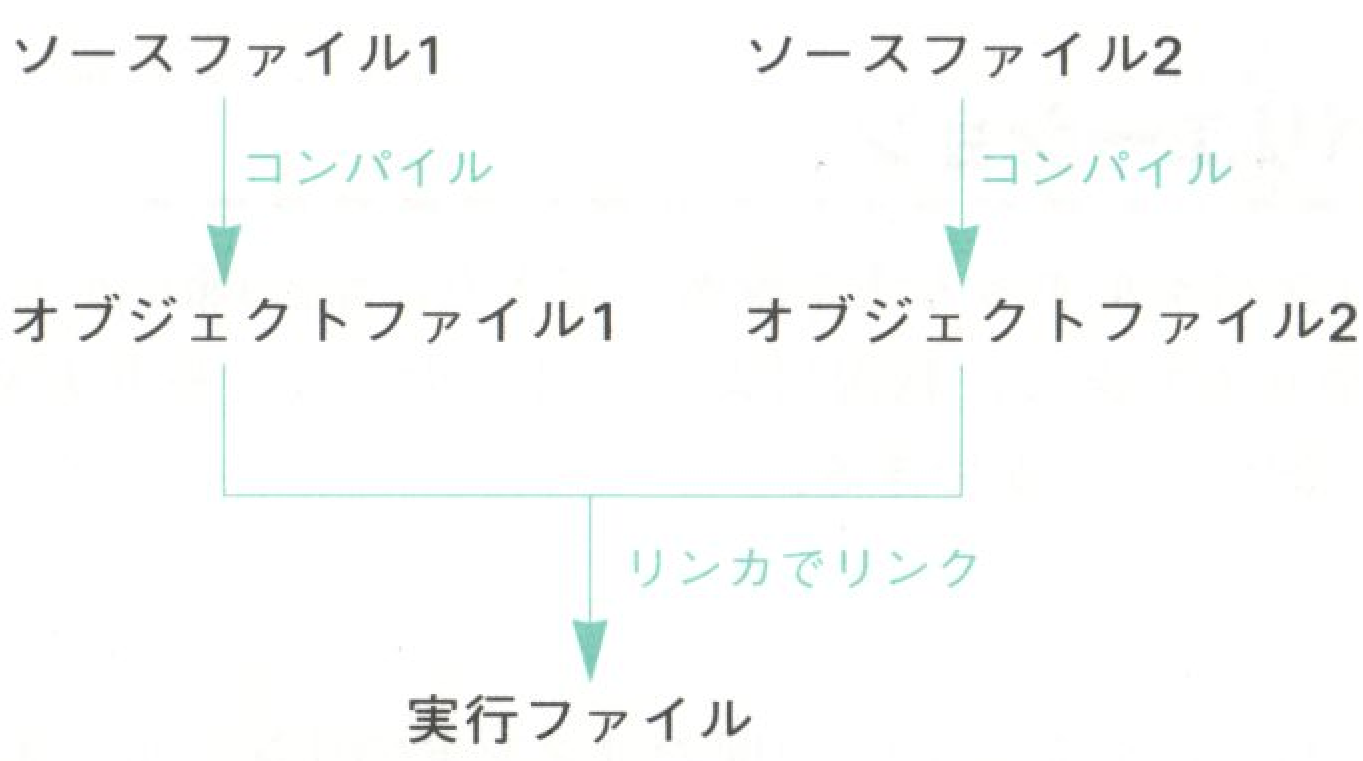
main.C → main.o (コンパイル)

util.C → util.o (コンパイル)

リンカは，複数のオブジェクトファイルをリンクすることができる．そこで，コンパイルされた2つのオブジェクトファイルをリンクして，実行ファイルを作成する．



これが，分割コンパイルのやり方である．次にソースファイルから実行ファイルまでの流れをまとめる．



2.3◆コンパイルとリンクのしかた

具体的なコンパイルのしかたは，次のようにする．

●オブジェクトファイル

ソースファイルをコンパイルしてオブジェクトファイルを作成する部分は，-cオプションを指定する．

```
g++ -c main.C ———— main.Cをコンパイルしてmain.oを作る
g++ -c util.C ———— util.Cをコンパイルしてutil.oを作る
```

-cオプションは，実行ファイルを作らないように指示するものである．

●実行ファイル

オブジェクトファイルをリンクして実行ファイルを作成する部分では，ソースファイルを指定する代わりにオブジェクトファイルを指定する．

```
g++ main.o util.o
```

――――main.oとutil.oをリンクしてa.outを作る

-oオプションを指定すると，a.outではなく，指定した名前で実行ファイルを作ることができる．

```
g++ -o go main.o util.o
```

――――main.oとutil.oをリンクする
実行ファイル名は“go”になる

2.4◆いろいろなバリエーション

ファイルを分割してコンパイルする方法を説明してきた．第2.3節で示した方法は，その最も基本的なやり方である．状況によっては，少し違うやり方もある．基本を踏まえた上で使い分けることができる．

●リンクまでを自動化

複数のファイルをコンパイルする場合も，1回のコマンドでリンクまで進めることができる．たとえば，src1.C，src2.C，src3.Cの3つのソースファイルをコンパイルし，リンクまで行うなら，次のようにする．

```
g++ src1.C src2.C src3.C
```

この結果，エラーがなければa.outという実行ファイルが作られる．このとき，途中で作られるオブジェクトファイルは削除される．もし，実行ファイル名をa.outではなく，指定したい場合は，次のようにする．

```
g++ -o go src1.C src2.C src3.C
```

↑ 実行ファイル名を指定（-o go）

●リンクをしない

複数のファイルを指定しても，リンクをしないで，オブジェクトファイルを作

るところでとどめることもできる．

```
g++ -c src1.C src2.C src3.C
```

↑ リンクをしない

この場合はa.outは作られない．オブジェクトファイルは削除されないので，src1.o，src2.o，src3.oは残る．

●ソースファイルとオブジェクトファイルの混在

3つのソースファイルのうち，src2.Cだけエラーになったとする．その場合は，src2.Cを修正してコンパイルをやり直すことになる．ただし，src1.Cとsrc3.Cは，コンパイルし直す必要がなく，オブジェクトファイルのsrc1.oとsrc3.oをそのまま使うことができる．つまり，次のようなコンパイルが可能である．

```
g++ src1.o src2.C src3.o
```

ここで指定したファイルは，ソースファイルとオブジェクトファイルが混在している．このような指定でも，コンパイルが必要なファイルはコンパイルされ，正しくリンクされる．

3 extern宣言

コンパイルのしかたを覚えただけでは，分割コンパイルはできない．まだ，知らなければならないことがいくつかある．この節では，extern宣言について説明する．

3.1◆コンパイルエラー

具体的に分割コンパイルを実践する．例として，第1節で取り上げたswap関数とその利用を使う．まず，次のutil.Cに関数swapの定義を入れる．

【util.C】

```
// xとyの値を交換する（第3版）
void swap(int &x, int &y)
```

```
{
    x = y - x;
    y -= x;
    x += y;
}
```

そして，関数swapを使用するコードを次のmain.Cに入れる．

【main.C】

```
#include <iostream.h>

int main()
{
    int n1 = 100, n2 = 200;

    cout << "n1, n2の最初の値: " << n1 << ", " << n2 << endl;
    swap(n1, n2);
    cout << "swap実行後の値　: " << n1 << ", " << n2 << endl;

    return 0;
}
```

util.Cの方は何の問題もなくコンパイルできる．

図27-1▼

```
% g++ -c util.C
%ls -l util*
-rw-rw-r--   1 kt       kt              101 Sep  2 09:55 util.C
-rw-rw-r--   1 kt       kt              755 Sep  2 16:05 util.o
% _
```

コンパイルの結果，util.oが作成されているのがわかる．util.oには，コンパイルされたswapのコードが入っている．util.oをリンクしたプログラムは，swapを使うことができる．

ここまでは順調である．しかし，次のmain.Cをコンパイルすると，エラーが発生する．

図27-2▼

```
% g++ -c main.C
main.C: In function `int main()':
main.C:10: warning: implicit declaration of function `int
swap(...)'
% _
```

分割コンパイルまでの道は険しい.

3.2◆エラーの原因

コンパイルは，ソースファイル単位で行われる．コンパイラは，コンパイル中のファイルだけを見てコンパイルを行う．他のソースファイルのことには無関心である.

ファイルmain.Cでは，関数mainの中でswapを使っている.

```
int main()
{
    ............
    swap(n1, n2);  ←―――――― ここ
    ............
}
```

そこで，コンパイラはswapを呼び出すコードを作ろうとする．しかし，ファイルの終わりまで探してもswapの定義が見つからない．swapの定義は別のファイルutil.Cの方にあるのだから当然である.

この結果，コンパイラは「swapが見つからないよ」といってエラーを報告する．これがエラーの原因である.

このエラーを取り除くには，swapの定義をmain.Cに移さなければならない．しかし，それでは分割コンパイルにならない．では，どうするかというと，次に説明するextern宣言を使うのである.

3.3◆extern宣言

第19章第3.3節で「定義と宣言」について説明した．変数や関数など名前の付い

たものには，定義と宣言がある．定義がその実体で，宣言はその名前の意味をコンパイラに教えるだけの仮のものである．たとえば次は定義であり，実体が作られる

```
int x;  ―――― 変数xの実体が作られる

int func(char *s)  ―――― 関数funcの実体が作られる
{
    ............
}
```

多くの宣言は定義を兼ねていて，宣言するだけでその実体が作られる．しかし，宣言を宣言だけにとどめ，定義を行わないようにすることもできる．それが，extern宣言である．

extern宣言は，その宣言が定義宣言ではないことを示す．その宣言の定義がどこか別のところで行われていることを示す．extern宣言を使えば，同じ名前の宣言を何度でも行えるが，次の条件を守らなければならない．

- その名前の使い方は正確に一致していなければならない
- どこかに定義宣言がなければならない

3.4◆変数に対するextern宣言

extern宣言は，変数に対するものと，関数に対するものがある．変数の場合は，宣言の前に「extern」と付ければよい．たとえば，次は普通の定義宣言である．

```
int gvar;  ―――― これは定義宣言
```

この場合は，変数gvarがメモリに割り当てられる．しかし，この宣言の前にexternというキーワードを付ければextern宣言になる．

```
extern int gvar;  ―――― これは宣言だけ
```

この場合，gvarという変数は作られない．ただ，gvarという名前がint型の変数であることを示すだけで，その本当の定義はどこか別の場所（同じソースコード内でも別のソースコードでもよい）にあることを示す．

初期値のある変数宣言をextern宣言するには注意が必要である．たとえば，次の

定義宣言は，初期値を指定している．

```
int gvar = 100;
```

この場合は，変数gvarがメモリに割り当てられ，その初期値の100が代入される．この定義宣言をextern宣言にするには，代入部分を取り除かなければならない．

```
extern int gvar = 100;  ──── これはダメ
extern int gvar;  ──── これならよい
```

3.5◆関数に対するextern宣言

関数をextern宣言するには，その関数のヘッダだけを取り出し，先頭に「extern」を付け，最後にセミコロンを付ける．

```
extern 関数ヘッダ;
```

たとえば次は，関数swapの定義宣言である．

```
void swap(int &x, int &y)
{
    x = y - x;
    y -= x;
    x += y;
}
```

この宣言により，関数swapのコードが作られる．この関数のextern宣言を作るには，次のようにする．

```
extern void swap(int &x, int &y);
```

この場合，swapという変数は作られない．このextern宣言により，コンパイラは，swapという名前が関数の名前であることを知り，その型や引数の形式を知ることができる．

3.6◆エラーの修正

extern宣言がわかると，第3.1節のエラーを取り除く方法がわかる．すなわち，main.Cにswapのextern宣言を加えればよい．

【main.C】

```
#include <iostream.h>

extern void swap(int &x, int &y); ―――――― 追加する

int main()
{
    int n1 = 100, n2 = 200;

    cout << "n1, n2の最初の値 : " << n1 << ", " << n2 << endl;
    swap(n1, n2);
    cout << "swap実行後の値  : " << n1 << ", " << n2 << endl;

    return 0;
}
```

これで，main.Cのコンパイルに成功することができる．

図27-3▼

```
% g++ -c main.C ―――――――――――――― main.Cをコンパイル
% _                                  エラーにならない
```

2つのソースファイルのコンパイルに成功した．あとは2つのオブジェクトファイルが作られていることを確かめ，それらをリンクすれば，実行ファイルを作成することができる．

図27-4▼

```
% ls *.o ――――――――――――――――――― オブジェクトファイルを確認
main.o  util.o
% g++ main.o util.o ――――――――――――― リンクする
% a.out ―――――――――――――――――――― 実行する
n1, n2の最初の値 : 100, 200
swap実行後の値   : 200, 100
100
% _
```

以上が，分割コンパイルの基本である．

4 ヘッダファイル

別のファイルにある定義（関数や変数など）を使うには，extern宣言が必要である．しかし，そのたびにextern宣言を書くのは煩わしいし，また間違いも発生しやすい．

#include疑似命令を使うと，extern宣言を書かなくても，別ファイルにある定義を利用することができる．

4.1◆#include疑似命令

疑似命令というのは，コンパイラではなく，プリプロセッサに対する命令である．

一般にコンパイラが実行される前に**プリプロセッサ**と呼ばれるコマンドが実行され，ソースファイルの変形を行う．プリプロセッサの実行は自動的に行われるので，通常，プログラマーに意識されることはない．

そのプリプロセッサに与える命令が疑似命令である．プリプロセッサは，ソースファイル中に現れる疑似命令を，コンパイラが理解できる形に変形する．

#includeは，疑似命令の1つである．その書式は，次のように#includeの後ろにファイル名を" "で囲んで指定する．

```
#include "filename"
```

この疑似命令が現れると，プリプロセッサは，指定されたファイルに書かれている内容をその位置に挿入する．取り込むファイルの名前は何でもよい．また，書かれている内容も何でもよい．

たとえば，otherというファイルの内容が以下のようなものとする．

```
Did you have trouble finding the place?
No, I didn't.
```

また，go.Cというファイルの内容が以下のようなものとする．

```
one
two
#include "other" ―――― この行に注目
three
```

この状態でgo.Cをコンパイルする．すると，コンパイルの直前にプリプロセッサが実行される．そして，その位置にotherの内容が挿入され，go.Cの内容が以下のように変わる．

```
one
two
Did you have trouble finding the place?
No, I didn't.
three
```

（"Did you have trouble finding the place?"の行に注記：otherの内容が挿入される）

これが，#includeの処理内容である．繰り返すと，#include疑似命令は，指定されたファイルの内容をその位置に挿入する．

4.2◆ヘッダファイル

通常，#includeはソースファイルの頭（*head*）の部分に書く．そこで，#includeで指定するファイルを**ヘッダファイル**（*headerfile*）という．

```
#include "filename"
```

（"filename"への注記：ヘッダファイル）

ヘッダファイルのファイル名は何でもよいが，通常は拡張子に“.h”を指定する．

```
#include "filename.h"
```

（"h"への注記：拡張子は“.h”）

4.3◆ヘッダファイルに書く内容

ヘッダファイルに書く内容は自由である．しかし，ヘッダファイルはC++のソースファイルに挿入される．C++のプログラムとして意味のあるものでなければ，その内容を挿入してもコンパイルエラーになってしまう．

では，C++の文法に適ったものなら何でもよいかというと，そうでもない．次のようなものは含めてはいけない．

関数定義

変数宣言
メンバー関数の本体の定義

ヘッダファイルには，機械語のコードには変換されない宣言だけを入れる．

変数のextern宣言
関数のextern宣言
インライン関数の宣言
class宣言
…………

もちろん，その規則を守らなくても文法的には間違いではない．しかし，守ったほうがベターである．その理由を次に説明する．

4.4◆分割コンパイルの支援

そもそもヘッダファイルは，分割コンパイルを支援するために存在する．たとえば，第3節のutil.Cを利用するプログラムは，次のextern宣言を書かなければならない．

```
extern void swap(int &x, int &y);
```

もし，ほかにも関数が入っていれば，その宣言も必要である．そのようなextern宣言を長々と書くのは煩わしい．そこで，util.hというヘッダファイルを作る．util.hにはutil.Cに入っている関数や，もしあればグローバル変数のextern宣言などを入れる．

util.hの内容

```
extern void swap(int &x, int &y);
その他の宣言
```

このようなutil.hを作っておけば，これをインクルードするだけでutil.Cを利用することができる．長々とextern宣言を書いたり，書き誤ることもなくなる．

では，もしutil.hに関数や変数の定義宣言の本体が入っていたらどうなるか？

グローバル定義（関数定義やグローバル変数の定義）は，プログラムの中で1つしか存在できない．つまり，同じ名前の関数やグローバル変数を2回定義することはできない．もし，util.hにswap関数本体の定義が入っていると，util.hをインクルードしたファイルとutil.Cで同じ定義が現れてしまう．また，util.hを取り込める

ファイルは，プログラム全体で1つだけに限定されてしまう．
ヘッダファイルに定義を入れてはいけない理由は，ここにある．

4.5◆条件コンパイル

ヘッダファイルに書く内容（書いてもよい内容）には，いくつか例外がある．たとえば，定数宣言がそうである．

```
const float PI = 3.14159;
```

このような定数は汎用的に使われるので，ヘッダファイルに含めてよい．そのヘッダファイルをインクルードするだけでPIの定義を利用できる．
ところが，定数宣言は定義宣言でもある．つまり，2回以上同じ宣言をしてはいけない．すると，そのヘッダファイルは1つのプログラムで1回しかインクルードすることができない．

そこで，#ifndefと#endifを使う．これらは，第4.1節にも登場した疑似命令の1つである．すなわち，プリプロセッサによって処理される．
#ifndefと#endifの使い方は，これらの疑似命令でソースファイルの一部または全部を挟む．

```
#ifndef 記号定数
............ ――――――――――――――― (1)
............ ――――――――――――――― (2)
............ ――――――――――――――― (3)
#endif
```

もし指定した記号定数が定義されていなければ，これらの疑似命令は意味を持たない．普通にコンパイルされる．しかし，もし記号定数が定義されていたなら，これらの疑似命令で挟まれた部分――すなわち，上のソースコードなら（1）～（3）に書かれている内容は，コンパイルされない．
つまり，#ifndefと#endif疑似命令を使うと，条件によってコンパイルしたり，しなかったりを制御することができるのである．これを**条件コンパイル**という．

4.6◆重複includeの回避

ヘッダファイルを作るとき，#ifndefと#endif，および記号定数を定義する

#defineという疑似命令を使う．まず，ヘッダファイルの先頭に次の2行を入れる．

```
#ifndef UTIL
#define UTIL
```

ここで指定したUTILは，**記号定数**である．記号定数というのは，プリプロセッサだけが認識する名前である．ここで指定する記号定数の名前は，何でもよい．しかし，ヘッダファイルと同じ名前（この例ではutil）で，大文字にしておくと，ほかの名前とバッティングしなくてよい．

そして，ヘッダファイルの最後に次の1行を入れる．

```
#endif
```

util.hの場合なら次のようになる．

```
#ifndef UTIL ―――――――――――――――――― (1)
#define UTIL ―――――――――――――――――― (2)

void swap(int &x, int &y); ―――――――― (3)

#endif ―――――――――――――――――――――― (4)
```

ヘッダファイルをこのように作ることによってどうなるか？　まず，そのプログラムで初めてutil.hがインクルードされたときを考える．この状態では，まだUTILは定義されていない．したがって，#ifndefは無視され，その本文（この場合は（3）の1行だけだが）がソースファイルに挿入される．また，（2）により，記号定数のUTILが定義される．

2回目以後にutil.hがインクルードされると，（2）により，すでにUTILが定義されている．このため，#ifndefにより（2），（3）の本体はスキップされ，ソースファイルに挿入されない（コンパイルされない）．つまり，1つのプログラムでutil.hが何回かインクルードされても悪影響を及ぼさない．

このように，ヘッダファイルに疑似命令を取り入れることにより，重複してインクルードされる問題を回避することができる．

5 標準ヘッダファイル

第4節では#includeを書くとき，ヘッダファイルを""で囲むように説明した．ところが，これまで書いてきたプログラムは，streamio.hなどのヘッダファイルを<>で囲んできた．

#include "filename" ―― 第4節の書き方
#include <iostream.h> ―― これまでのプログラムの書き方

ヘッダファイルを囲む""と<>の違いは何か？
ヘッダファイルを指定する部分は，囲み記号によって2通りの書き方がある．

#include <filename.h> ―― 専用ディレクトリから入力する
#include "filename.h" ―― カレントディレクトリから入力する

ヘッダファイルが<>で囲まれていると，コンパイラは，そのファイルが専用のディレクトリ（/usr/includeなど）にあるものと見なす．したがって，そのヘッダファイルはヘッダファイル専用のディレクトリから読み込まれる．

ヘッダファイルが""で囲まれていると，そのヘッダファイルはカレントディレクトリから読み込まれる．

ヘッダファイルには，iostream.hのように最初から用意されているものと，ユーザーが作成するものがある．前者の標準的なヘッダファイルは，専用のディレクトリにファイルされている．したがって，<>で囲む記法は，コンパイラが用意した標準のヘッダファイルを入力するのに使用する．""で囲む記法は，ユーザーが作成したヘッダファイルを入力するのに使用する．

6 やさしいmake

分割コンパイルを行う利点の1つに，プログラムの修正などで再コンパイルするとき，必要なファイルだけを再コンパイルすればよいので開発時間を短縮できるということがある．しかし，分割するファイルの数が増え，**ファイルの依存関係**が複雑になってくると，再コンパイルが必要なファイルの選択が難しくなってくる．せっかくプログラムを修正しても，必要なファイルがすべて再コンパイルさ

れていないため，動かないということも起こる．

makeというコマンドを使うと，分割コンパイルを自動化することができる．再コンパイル時におけるファイルの選択を自動的に行うことができる．

make自体は汎用的なツールで，分割コンパイルだけを目的としたものではない．高度なバッチ処理コマンドと見ることができる．しかし，makeが分割コンパイルに威力を発揮することは間違いない．

この節では，分割コンパイルを例にmakeの基本を説明する．

6.1◆ファイルの依存関係

最初に，分割コンパイルにおける**ファイルの依存関係**を説明する．

いま，あるプログラムが3つのソースファイルでできているとする．

console.C ―――――― キーボードや画面関係の処理
window.C ―――――― ウィンドウ関係の処理
main.C ―――――― メインプログラム

console.Cは，console.hというヘッダファイルをインクルードしている．

console.h → console.C

この場合，「console.Cはconsole.hに依存している」という．consol.Cは，自分自身が修正された場合だけでなく，console.hが変更されたときにも再コンパイルしなければならない．

次に，window.Cはwindow.hをインクルードしている．また，console.Cの機能を利用しているため，console.hもインクルードしている．

console.h，window.h → window.C

この場合，「window.Cはconsole.hとwindow.hに依存している」という．window.Cは，自分自身が修正された場合だけでなく，console.hやwindow.hが変更されたときにも再コンパイルしなければならない．

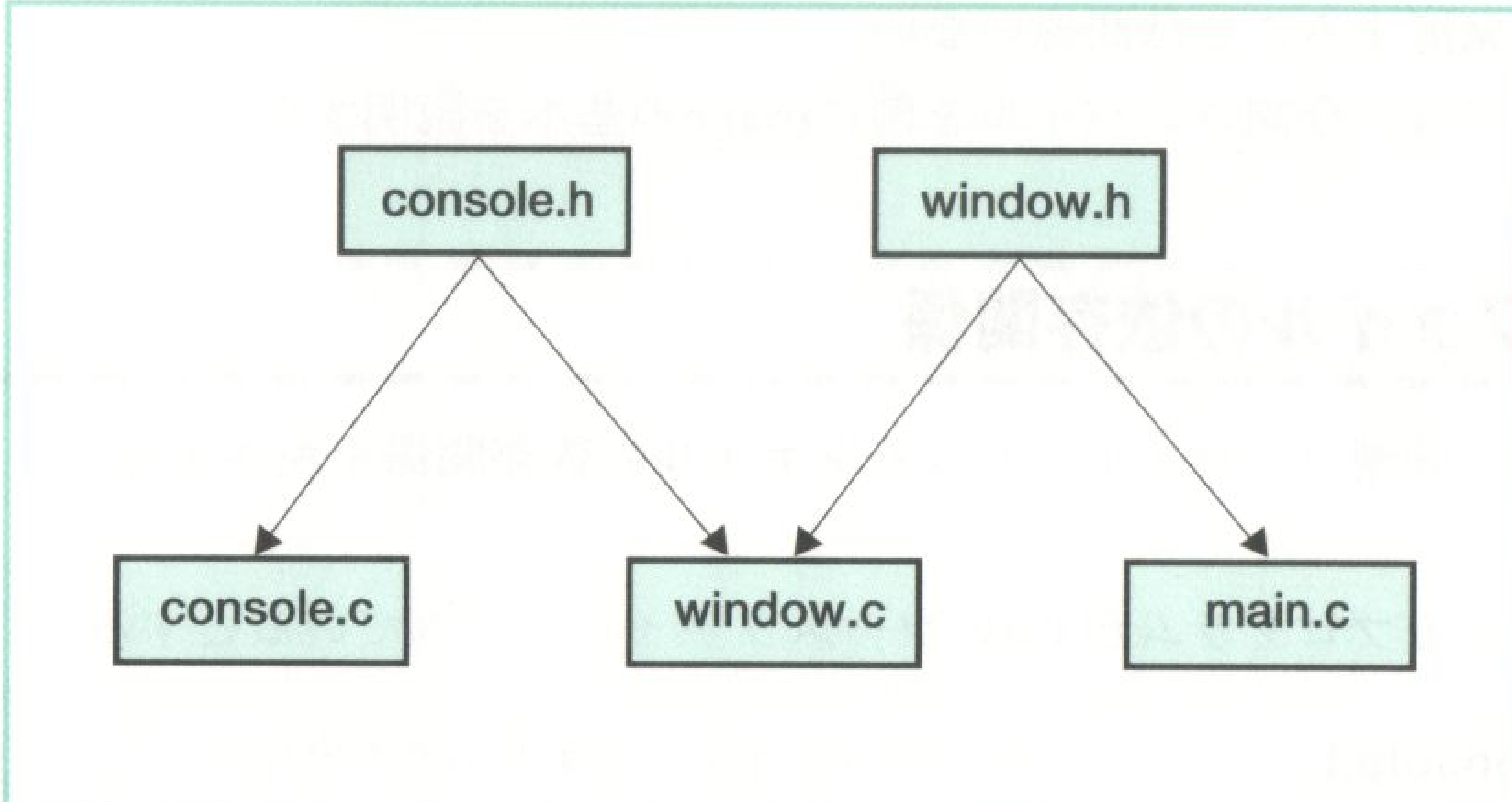
最後に，main.Cは，window.Cの機能を利用しているため，window.hをインクルードしている．

window.h → main.C

この場合，「main.Cはwindow.hに依存している」という．main.Cは，自分自身が修正された場合だけでなく，window.hが変更されたときにも再コンパイルしなければならない．

以上，このプログラムにおけるファイルの依存関係をまとめると次のようになる．

図27-5▶



これらのプログラムを初めてコンパイルするときは，もちろん3つのソースファイルをすべてコンパイルしなければならない．

その後，main.Cを修正したとする．その場合は，すべてのファイルをコンパイルし直す必要はない．main.Cだけを再コンパイルし，リンクし直せばよい．同様に，console.Cを修正した場合は，console.Cだけを再コンパイルすればよい．

間違いやすいのは，console.hを修正した場合である．cosole.Cを再コンパイルしなければならないのは当然として，window.Cの再コンパイルも忘れてはいけない．window.Cもconsole.hをインクルードしているからである．

この例では，ソースファイルが3つしかなく，ファイルの間の依存関係も単純なので，再コンパイルも単純である．しかし，ファイルの数が増え，ファイルどうしの依存関係が複雑になってくると，再コンパイルが難しくなってくる．

makeというコマンドを使うと，分割コンパイルを自動化することができる．再コンパイル時におけるファイルの選択も自動的に行われる．次に，makeを使うとコンパイルや再コンパイルがどれだけ楽になるかを実際に示す．

6.2◆makeの実際

makeコマンドを具体的に投入する．例として，この章で取り上げてきたmain.Cとutil.Cを使う．各ファイルの依存関係は次のようになっている．

util.h → main.C
util.C（特に依存しているソースファイルはない）

初めてコンパイルするときは，ただ「make Enter」と入力するだけでよい．次に，その実行の様子を示す．

図27-6▼

```
% make
g++ -c main.C ―――――――――――― main.Cをコンパイル
g++ -c util.C ―――――――――――― util.Cをコンパイル
g++ main.o util.o ――――――――― main.oとutil.oをリンク
% _
```

すべてのソースファイルがコンパイルされ，リンクされているのがわかる．

次にutil.Cを修正したとする．しかし，今度も，ただ「make Enter」と入力するだけでよい．

図27-7▼

```
% make
g++ -c util.C ―――――――――――― util.Cをコンパイル
g++ main.o util.o ――――――――― main.oとutil.oをリンク
% _
```

main.Cのコンパイルは省略されているのがわかる．必要なファイルだけが再コンパイルされ，リンクされている．

次にutil.hを修正する．今度は，少しややこしい．どのファイルを再コンパイルすればよいだろうか？――しかし，これも考えることはない．正しい解をmakeが教えてくれる．ただ，「make Enter」と入力するだけでよいのである．

図27-8▼

```
% make
g++ -c main.C ―――――――― main.Cをコンパイル
g++ main.o util.o ―――――― main.oとutil.oをリンク
% _
```

main.Cはutil.hをリンクしているので再コンパイルが必要である．だから，コンパイルされている．しかし，util.Cはutil.hをインクルードしていないので再コンパイルは不要である．だから，コンパイルが省略されている．

次にa.outを削除するとどうなるか？　ソースファイルは，もとのままである．しかし，実行ファイルはなくなっている．この場合でも，ただ「make Enter」と入力するだけでよいのである．

図27-9▼

```
% make
g++ main.o util.o ―――――― リンクだけが実行される
% _
```

これを見ると，リンクだけが実行されている．コンパイルは行われていない．このように，どのような場合でも「make Enter」と入力するだけで必要最低限の処理だけをしてくれる．これが，分割コンパイルにおけるmakeの威力である．

6.3◆メイクファイル

どのような場合でも「make Enter」と入力するだけで，必要最低限の処理だけをしてくれるのは，makeが何をしてほしいかを知っているからである．逆にいえば，何をしてほしいかをmakeに教えてやらなければ，makeはすべきことがわからない．次のようにエラーを報告してくる．

図27-10▼

```
% make ―――――――――――――― makeに何も教えていない場合
make: *** No targets specified and no makefile found.  Stop.
%
```

この実行例では，makeが「makefileが見つからないよ」と報告している．

makeにすべきことを教えるには，**メイクファイル**（*makefile*）という，makeのための定義ファイルを作る．そして，その中に何をしてほしいかを記述する．

メイクファイルのファイル名は何でもよい．ただし，次のいずれかの名前にすると，makeの実行が楽になる（通常は，makefileを使う）．

```
GNUmakefile
makefile
Makefile
```

注意　これは，GNUのmakeの場合である．

カレントディレクトリにこれらのいずれかのファイルがあると，makeはそれをメイクファイルとして，その内容に従って処理を行う．つまり「make Enter」と入力するだけで実行できる．

しかし，これ以外のファイル名にした場合は，そのファイル名を-fオプションを使って明示的に指定する必要がある．

```
make -f file
```

↑ メイクファイル名（file）

なお，GNUmakefile，makefile，Makefileのファイルが複数個ある場合は，GNUmakefile，makefile，Makefileの順に検索される．

6.4◆ルール

メイクファイルには，**ルール**という単位で指示を書き込んでいく．ルールは，次のようなことを記述したものである．

- どんなファイルを作りたいのか
- そのファイルは，どのファイルに依存しているか
- そのファイルを作るには，どのようにコマンドを実行するか

ルールを書く順序は，一般的なプログラムとは逆になる．一般的なプログラムでは，実行する順に命令を並べていく．

命令1
命令2
命令3
……

しかし，メイクファイルの場合は最初に目的を書き，次にその目的を実現する手段を書いていくという順序になる．

最終目的
その目的を実現するための目的
その目的を実現するための目的
…………

6.5◆依存記述行

たとえばmain.Cとutil.Cをコンパイルして，実行ファイルa.outという名前の実行ファイルを作りたいとする．a.outが最終目標なので，まずa.outと書く．

a.out ――――― 最終目標

次にセミコロンで区切って，そのファイルがどのファイルに依存しているかを書く．

依存ファイルが複数あるときは，スペースで区切る．

a.out : main.o util.o

こうして出来上がった行を**依存記述行**という．依存記述行の先頭に書かれる目標ファイルのことを**ターゲット**という．また，コロンの右に書かれるファイルのことを**依存ファイル**という．



以上をまとめると，依存記述行の書式は次のようになる．

ターゲット:依存ファイル 依存ファイル …

6.6◆コマンド行

依存記述行の下にターゲットの作り方を記述する．要するに，ターゲットを作るためのコマンドを書く．たとえば，実行ファイルa.outを作るためのコマンドは，次のようになる．

```
g++ main.o util.o
```
――― これを依存記述行の下に書く

これを**コマンド行**という．コマンド行は依存記述行と区別するため，先頭にタブを入れる．

```
a.out : main.o util.o
	g++ main.o util.o
```
a.out : main.o util.o ――― 依存記述行
g++ main.o util.o ――― コマンド行（先頭にタブがある）

必要なら複数のコマンド行を書くこともできる．

6.7◆基本的なメイクファイル

最終目標a.outについてのルールを記述したら，次にその一歩手前の目標を書く．a.outの依存ファイルは，main.oとutil.oである．したがって，main.oとutil.oについてのルールを書けばよい．

main.oについてのルール

```
main.o : main.C util.h
	g++ -c main.C
```

util.oについてのルール

```
util.o : util.C
	g++ -c util.C
```

さらに，これらのターゲットの依存ファイルについてのルールを作るわけであるが，main.C，util.h，util.Cのいずれも，どのファイルにも依存していないので，ルールは必要ない．すなわち，これで完了である．次に，main.Cとutil.Cをコンパイルするためのメイクファイルの完成版を示す．

図27-11▼

```
a.out : main.o util.o
        g++ main.o util.o

main.o : main.C util.h
        g++ -c main.C

util.o : util.C
        g++ -c util.C
```

全体が3つのルールで出来上がっている．最初が最終目標であるa.outの作り方についてのルールであり，main.oとutil.oについてのルールが続いている．

第28章

getoptパッケージ

この章では，コマンドラインで指定された引数をプログラムの中で利用する方法を扱う．

コマンドライン引数を取得するには，main関数の引数を参照するのが基本である．この方法を使えば，コマンドライン引数をどのようにでも処理することができる．逆に，このことが，コマンドごとに引数の構文が不統一になるという現象を生んでいる．

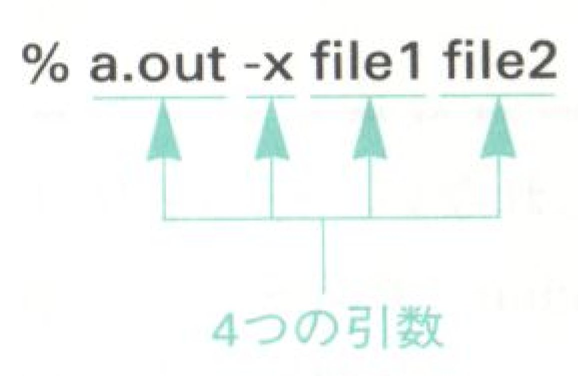
コマンドの構文を統一する1つの試みとして，getoptパッケージがある．getoptは，コマンドラインで指定されたオプションを解析する関数である．

1 コマンドラインの引数の解析

コマンドは，シェルプロンプトに対して行単位で入力する．その入力された1行の文字列を**コマンドライン**という．

% a.out -x file1 file2 ——— コマンドライン

C++から見た**引数**とは，コマンドラインで指定された文字列を空白で区切った個々の単語である．したがって，コマンドも引数ということになる．



引数には，3つの種類がある．

コマンド ―――― コマンドラインの先頭に置かれる引数
オプション ―――― マイナス-で始まる引数
非オプション ―――― マイナス-で始まらない引数

この節では，C++が提供するコマンドライン引数を取得する基本方法を扱う．getoptパッケージも，この基本方法を使ってインプリメントされている．

1.1◆mainの引数

これまでのプログラムでは，main関数を引数なしで使ってきた．

```
int main()
{
    ........................
}
```

main関数は，必要なら3つまでの引数を指定することができる．コマンドライン引数を取得するには，そのうちの最初の2つを使う．

第1引数の型 ―――― int
第2引数の型 ―――― char *の配列

したがって，mainの書式は次のようになる．

```
int main(int ac, char *av[])
```

もちろん，mainの引数名は何でもよい．しかし，一般的には第1引数にargc，第2引数にargvを使う．

```
int main(int argc, char *argv[])
```

本書では，略記した2文字のacとavを使う．

1.2◆引数の値

mainの第2引数の型char *av[]がわかりづらいかもしれない．その意味は，avが配列であることを示している．また，その要素の型がchar *であることを示している．char *は文字へのポインタでもある．したがって，avの各要素は文字列を指し

ていると考えてもよい．

```
av[0] ──────────── ある文字列の先頭アドレスを指す
av[1] ──────────── ある文字列の先頭アドレスを指す
av[2] ──────────── ある文字列の先頭アドレスを指す
………
```

そして，（多分，推察どおり）これらの配列要素には各コマンドライン引数が入る．

```
av[0] ──────────── コマンドラインで指定された第1引数を指す
av[1] ──────────── コマンドラインで指定された第2引数を指す
av[2] ──────────── コマンドラインで指定された第3引数を指す
………
```

それを次のサンプルプログラムで確かめる．

【サンプルプログラム】

これは，配列avの最初の4つの要素を表示するものである．推察が正しければ，コマンドライン引数が4つ分表示されるはずである．

```
#include <iostream.h>

int main(int ac, char *av[])
{
    cout << av[0] << endl ──────────── 第1引数を出力
         << av[1] << endl ──────────── 第2引数を出力
         << av[2] << endl ──────────── 第3引数を出力
         << av[3] << endl; ─────────── 第4引数を出力

    return 0;
}
```

次に，4つの引数を指定した場合の実行例を示す．

【実行例】

```
% a.out one two three four
a.out ──────────────────── av[0] の値
one ────────────────────── av[1] の値
two ────────────────────── av[2] の値
three ──────────────────── av[3] の値
% _
```

推察どおり，コマンドライン引数が表示されている．1つ注意するとすれば，av[0]には第1引数ではなく，コマンド名が入るということである．

この節の最初のところで指摘したように，C++はコマンド名を第1引数と見なす．これは，コマンド名といっしょに指定されたパス名を調べたり，コマンド名で処理内容を変更したりといったように，指定されたコマンドを取得できるようにするためである．

1.3◆引数の数

前項の実行例では，one two three fourの4つの引数を指定した．しかし，プログラムで表示したav[]の数が足りなかったので，最後の引数が表示されていない．

コマンドラインで指定される引数の数は，実行するたびに変わる．その全引数を調べるためには，実際に指定された引数の数が必要になる．

main関数の第1引数を使えば，指定された引数の数を取得することができる．

【サンプルプログラム】

これは，単純にacの値を表示するものである．

```
#include <iostream.h>

int main(int ac, char *av[])
{
    cout << ac << endl;

    return 0;
}
```

【実行例】

```
% a.out                          引数の指定なし
1
% a.out one two three            3つの引数を指定
4
% _
```

ここでは，2通りの実行例を示した．acの値が指定した引数の数よりも1大きい．これは，コマンド名も引数と数えれば，引数の数に一致する．

1.4◆全引数の取得

コマンドライン引数の指定が可能なプログラムでは，ループを使って全引数を取得するのが一般的である．そのとき，ループの回数をmainの第1引数で制御する．

```
for (int i = 0; i < ac; i++)
    // 引数av[i]を処理する
```

次にコマンドラインで指定された全引数を表示するプログラムを示す．

【サンプルプログラム】

```
#include <iostream.h>

int main(int ac, char *av[])
{
    for (int i = 0; i < ac; i++)
        cout << av[i] << endl;

    return 0;
}
```

【実行例】

```
% a.out one two three
a.out
one
two
three
% _
```

1.5◆解析が各プログラムに任されている

以上，C++が提供するコマンドライン引数を取得する方法を見てきた．C++のみならず，C言語においても同様の方法でコマンドライン引数を取得することができる．

UNIXにおいては，コマンドライン引数の解釈が，各コマンドに任されている．このため，コマンドライン引数の構文がコマンドごとに統一されていないという傾向がある．極端にいえば，オプションの先頭を - 以外の他の文字にすることも自由で

ある．たとえば，次のようにすると，オプションの先頭文字を@にすることができる．

```
for (int i = 0; i < ac; i++)
    if (*av[i] == '@') ———— 引数の先頭文字が@ならば
        // オプションと見なす
```

コマンドライン引数の構文を統一するための1つの試みとして，getopt関数がある．getoptについては，第3節で説明する．

2 環境変数の取得

第1.1節で，「main関数は必要なら3つまでの引数を指定できる」ことを説明した．main関数に第3引数を指定することで，環境変数を取得することができる．

main関数の第3引数の型はchar *の配列である．

```
int main(int ac, char *av[], char *ev[])
                             ↑
                           第3引数
```

推察のとおり，この配列が環境変数を指している．

```
ev[0] ———— 1つ目の環境変数
ev[1] ———— 2つ目の環境変数
ev[2] ———— 3つ目の環境変数
.........
```

コマンドライン引数のときは，第1引数がコマンドライン引数の数を表していた．環境変数の場合は，第1引数に相当するものはない．しかし，それ以上環境変数がない場合，配列要素の値はゼロになる．したがって，次のようなループを作れば，全環境変数を処理することができる

```
for (int i = 0; ev[i] !=0 ; i++)
    // ev[i]が次の環境変数を指している
```

あるいは「!= 0」の部分は省略できるので，次のように書いてもよい．

```
for (int i = 0; ev[i]; i++)
    // ev[i]が次の環境変数
```

【サンプルプログラム】

これは，全環境変数の値を表示するものである．

```
#include <iostream.h>

int main(int ac, char *av[], char *ev[])
{
    for (int i = 0; ev[i]; i++)
        cout << ev[i] << endl;

    return 0;
}
```

【実行例】

```
% a.out
HISTSIZE=1000
LOGNAME=kt
HISTFILESIZE=1000
MAIL=/var/spool/mail/kt
PAGER=less
............
% _
```

3 getopt

getoptは，コマンドライン引数のうち，オプションの解析を行う汎用関数である．getoptを使って作られたプログラムは，次のように統一された構文を持つことができる．

- 1文字オプションは - で始める
- 複数の1文字オプションを1つの - の後にまとめて指定することができる
- 引数付きのオプションを1つの - にまとめるときはいちばん最後に指定する

この節では，getopt関数の使い方を扱う．

3.1◆getoptの引数

getopt関数は，コマンドラインオプションを解析する．getopt関数を使用するときは，ヘッダファイルのunistd.hをインクルードしなければならない．

```
#include <unistd.h>
```

getoptは3つの引数をとる．1つ目と2つ目の引数には，main関数の引数をそのまま渡す．たとえば，main関数を次のように書いたとする．

```
int main(int ac, char *av[])
```

この場合，getoptの実行は次のようになる．

```
getopt(ac, av,第3引数)
```

getoptの第3引数には，解析するオプション文字を並べた文字列を指定する．たとえば，そのプログラムが -x，-y，-zの3つのオプションを使用するなら "xyz" を指定する．

```
getopt(ac, av, "xyz");
```

3.2◆getoptの関数値

getopt関数を実行すると，関数値としてコマンドラインで指定されたオプション文字を，指定された順序で1つずつ返す．たとえば，次のように-xと-yの2つのオプションが指定されたとする．

```
% a.out -x -y
```

この場合，getoptを実行するたびにxとyの文字が返される．

第1回目のgetoptの関数値= 'x'
第2回目のgetoptの関数値= 'y'

それ以上オプションがないときは，getoptは-1を返す．

次のサンプルプログラムで，その具体的な使い方を示す.

【サンプルプログラム】

これはgetoptをwhileループの中で使うやり方を示すものである．case文でどのオプションが指定されたかを区別している.

```
#include <iostream.h>
#include <unistd.h>

int main(int ac, char *av[])
{
    int c;

    while ((c = getopt(ac, av, "xyz")) != -1)
        switch (c) {
        case 'x':
            cout << "-xオプションが指定された" << endl;
            break;
        case 'y':
            cout << "-yオプションが指定された" << endl;
            break;
        case 'z':
            cout << "-zオプションが指定された" << endl;
        }

    return 0;
}
```

【実行例】

```
% a.out -xyz
-xオプションが指定された
-yオプションが指定された
-zオプションが指定された
% _
```

3.3◆無効なオプション文字

コマンドラインで無効なオプション文字が指定されると，getoptは，次のようなエラーメッセージを表示する.

a.out: illegal option -- a

実行ファイル名　　無効なオプション文字

たとえば，前の節のプログラムで無効なオプション文字aとbを指定すると，次のようなエラーメッセージが表示される．

図28-1▼

```
% a.out -ab
a.out: illegal option -- a
a.out: illegal option -- b
% _
```

getoptが報告するエラーメッセージは，標準エラー出力に出力される．

図28-2▼

```
% a.out -xabyz 2> error ―――― エラーメッセージをerrorにリダイレクト
-xオプションが指定された
-yオプションが指定された
-zオプションが指定された
% cat error ―――― errorの内容を見る
a.out: illegal option -- a
a.out: illegal option -- b
% _
```

3.4◆使い方の表示

無効なオプション文字が指定されたとき，そのコマンドの使い方を表示させたいとする．getoptは，無効なオプション文字が指定されると，文字？を返す．そこで，解析ループのcase文に '?' のセクションを追加し，そこに使い方を表示するコードを書く．

```
while ((c = getopt(ac, av, "xyz")) != -1)
      switch (c) {
      ............
      case '?':
```

```
        cout << "usage: " << av[0] << " -xyz" << endl;
    }
```

【サンプルプログラム】

これは，無効なオプション文字が指定されたとき，コマンドの使い方を表示するプログラムである．有効なオプション文字はx，y，zの3文字である．

```
#include <iostream.h>
#include <unistd.h>

int main(int ac, char *av[])
{
    int c;

    while ((c = getopt(ac, av, "xyz")) != -1)
        switch (c) {
        case 'x':
            cout << "-xオプションが指定された" << endl;
            break;
        case 'y':
            cout << "-yオプションが指定された" << endl;
            break;
        case 'z':
            cout << "-zオプションが指定された" << endl;
            break;
        case '?':
            cout << "usage: " << av[0] << " -xyz" << endl;
    }

    return 0;
}
```

【実行例】

```
% a.out -a ―――――――――― aは無効なオプション文字
a.out: illegal option -- a
usage: a.out -xyz
% _
```

case '?' の代わりにdefaultを使ってもよい．

```
while ((c = getopt(ac, av, "xyz")) != -1)
    switch (c) {
    …………
    default:
        cout << "usage: " << av[0] << " -xyz" << endl;
}
```

3.5◆エラーメッセージの停止 —— opterr

無効なオプション文字が指定されると，getoptは無条件でエラーメッセージを表示する．

```
a.out: illegal option -- a
```

このようなエラーメッセージの代わりに独自のエラーメッセージを作って表示することもできる．そのためには，標準で表示されるエラーメッセージを表示しないように設定する必要がある．

getoptを使用するときunistd.hをインクルードするが，このヘッダファイルをインクルードすると**opterr**というグローバル変数を使えるようになる．opterrを0に設定すると，無効なオプション文字が指定されてもエラーメッセージを表示しなくなる．

```
opterr = 0;
```

【サンプルプログラム】

```
#include <iostream.h>
#include <unistd.h>

int main(int ac, char *av[])
{
    int c;

    opterr = 0;———————————————追加する

    while ((c = getopt(ac, av, "xyz")) != -1)
        switch (c) {
        case 'x':
            cout << "-xオプションが指定された" << endl;
```

```
            break;
        case 'y':
            cout << "-yオプションが指定された" << endl;
            break;
        case 'z':
            cout << "-zオプションが指定された" << endl;
            break;
        case '?':
            cout << "usage: " << av[0] << " -xyz" << endl;
        }

    return 0;
}
```

【実行例】

```
% a.out -a ―――――――――――――――――― 無効なオプション文字を指定
usage: a.out -xyz ―――――――――――――― エラーメッセージは表示されない
% _
```

今度は，コマンドの使い方だけで，エラーメッセージが表示されていない．

3.6◆引数をとるオプション

オプションの中には，引数をとるものがある．たとえば，-fの後ろにファイル名を指定したり，-nの後ろに数値を指定したりのように．

% a.out -f filename -n 1200

getoptは，引数をとるオプションを解析することもできる．引数はオプションに続けて入力する．そのとき，間を空白で区切っても区切らなくても正しく解釈する．

% a.out -f filename ―――― 空白で区切った

% a.out -ffilename ―――― 空白で区切らない

複数のオプションをまとめて指定する場合，引数をとるオプションはいちばん最後に指定する．すなわち，次はいずれも同じオプションである．

% a.out -xyzf filename

% a.out -x -y - z -f filename

getoptで引数をとるオプションを解析する場合は，第3引数に与えるオプション文字の集合で，オプション文字の次にコロンを置く．たとえば次は，fとnが引数をとるオプションである．

```
getopt(ac, av, "xyzf:n:")
```

getoptを使用するときunistd.hをインクルードするが，このヘッダファイルをインクルードするとoptargというグローバル変数を使えるようになる．

getoptが引数付きのオプションを検索すると，グローバル変数optargにその引数がセットされる．逆に引数をとらないオプションを解析すると，optargにはゼロ（空の文字列）がセットされる．なお，optargの型はchar *である．

```
extern char *optarg;
```

次に，引数付きのオプションを解析するサンプルプログラムを示す．

【サンプルプログラム】

これはx，y，z，f，nの5種類のオプションをとるプログラムである．そのうちのfとnは，引数付きのオプションである．

```
#include <iostream.h>
#include <unistd.h>

int main(int ac, char *av[])
{
    int c;

    while ((c = getopt(ac, av, "xyzf:n:")) != -1)
        switch (c) {
        case 'x':
            cout << "-xオプションが指定された" << endl;
            break;
        case 'y':
            cout << "-yオプションが指定された" << endl;
            break;
        case 'z':
            cout << "-zオプションが指定された" << endl;
            break;
        case 'f':
            cout << "-fオプション: " << optarg << endl;
            break;
        case 'n':
```

```
            cout << "-nオプション: " << optarg << endl;
            break;
        default:
            cout << "usage: " << av[0] << " -xyz" << endl;
    }

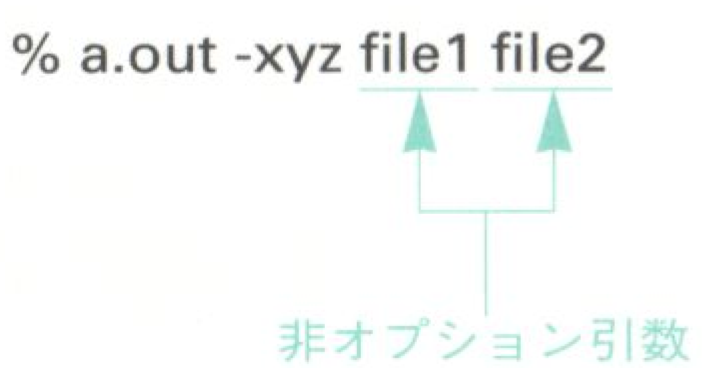
    return 0;
}
```

【実行例】

```
% a.out -xyzf filename -n 1200
-xオプションが指定された
-yオプションが指定された
-zオプションが指定された
-fオプション: filename
-nオプション: 1200
% _
```

3.7◆非オプション引数

ここまでオプションの解析を見てきたが，非オプション引数——すなわち，- で始まらない一般的な引数をどう処理すればよいかを説明する．



getoptを使用するときunistd.hをインクルードするが，このヘッダファイルをインクルードすると**optind**というグローバル変数を使えるようになる．

optindには，getoptが次に解析する配列av（mainの第2引数）のインデックスが入っている．初期値は1である．なぜ初期値が0でないかというと，av[0]にはコマンド名がセットされているので，引数の解析をav[1]から始めるからである．

getoptがオプションを1つ解析すると，optindは1つインクリメントされる．すなわち，av[optind]が次に解析する引数になる．したがって，すべてのオプションの解析が終わった直後のoptindが，非オプション引数の先頭を指している．

```
// このwhileループでオプションの解析を行う
while ((c = getopt(ac, av, "xyzf:n:")) != -1)
    switch (c) {
    ............

    }

// whileループを抜けた段階でav[optind]が
// 最初の非オプション引数になる
```

全引数の数はac（mainの第1引数）でわかる．したがって，次のようなループを作れば，すべての非オプション引数を処理することができる．

```
while (optind < ac)
    // av[optind++]が非オプション引数
```

【サンプルプログラム】

これは，getoptを使った引数の解析の集大成である．単純なオプションは-x，-y，-zの3つである．また，引数付きのオプションは-fと-nの2つである．最後に非オプション引数の一覧を表示する．

```
#include <iostream.h>
#include <unistd.h>

int main(int ac, char *av[])
{
    int c;

    // -で始まるオプションの解析
    while ((c = getopt(ac, av, "xyzf:n:")) != -1)
        switch (c) {
        case 'x':
            cout << "-xオプションが指定された" << endl;
            break;
        case 'y':
            cout << "-yオプションが指定された" << endl;
            break;
        case 'z':
            cout << "-zオプションが指定された" << endl;
            break;
        case 'f':
```

```
            cout << "-fオプション: " << optarg << endl;
            break;
        case 'n':
            cout << "-nオプション: " << optarg << endl;
            break;
        default:
            cout << "usage: " << av[0] << " -xyz" << endl;
        }

        // -で始まらない非オプションの表示
        while (optind < ac)
            cout << "ファイル" << av[optind++]
                 << "を処理します" << endl;

    return 0;
}
```

【実行例】

```
% a.out -xyz -f filename -n 1200 file1 file2 file3
-xオプションが指定された
-yオプションが指定された
-zオプションが指定された
-fオプション: filename
-nオプション: 1200
ファイルfile1を処理します
ファイルfile2を処理します
ファイルfile3を処理します
% _
```

第29章 乱数

乱数（*random number*）を簡単にいうと，でたらめな順序に並んだ数である．といっても，ただでたらめなだけではなく，どの区間をとっても各数の出現頻度が同じにならなければならない．

プログラムでは，よく乱数が使われる．統計の実験に使ったり，ゲームに使ったり，シミュレーション（模擬実験）に使ったりする．

randという関数を使うと，プログラムの中で乱数を発生することができる．ただし，randは計算によって乱数を作っている．したがって，randによって作られる乱数は本物の乱数ではなく，**疑似乱数**である．

疑似乱数 ―――― 本物の乱数ではないが
実用レベルで乱数の代わりになる

この章では，rand関数の使い方を説明する．

1 rand

乱数を発生させるには，randという関数を使う．この関数は，すでにライブラリとして定義されているので，自分で定義する必要はない．

1.1◆stdlib.h

randを使うには，stdlib.hをインクルードする．

```
#include <stdlib.h>
```

randの書式は，次のようになっている．

```
int rand();
```

引数はない．関数値はintである．関数randを実行するたびに新しい乱数が返される．

```
rand(); ——————— 1つ目の乱数を発生する
rand() ——————— 2つ目の乱数を発生する
rand(); ——————— 3つ目の乱数を発生する
…………
```

1.2◆乱数発生プログラム

関数randを使って実際に乱数を発生させるプログラムを次に示す．

【サンプルプログラム】

これは，randを使って7個の乱数を発生するプログラムである．

```
#include <iostream.h>
#include <stdlib.h> ——————— randを使うために必要

int main()
{
    int r;

    for (int i = 0; i < 7; i++) ——————— forループで7回繰り返す
        cout << rand() << endl; ——————— ここで乱数を発生している

    return 0;
}
```

プログラムのポイントを説明する．まず，関数randを使うため，stdlib.hをインクルードする必要がある．

```
#include <stdlib.h>
```

乱数を7個発生するため，forループを使う．

```
for (int i = 0; i < 7; i++)
    // ここで乱数を発生する
```

実際に乱数を発生しているのは，次の文である．

```
cout << rand() << endl;
```

ここで乱数を発生している

次に実行例を示す．

【実行例】

```
% a.out
269167349            乱数が7個表示されている
1169529124
889801541
1254073978
374297457
2065258565
1482041942
% _
```

2 乱数系列

第1.2節で7つの乱数を発生した．一見，でたらめな数が発生したように見える．しかし，ここに少し問題がある．プログラムを何回実行しても，同じ乱数が発生してしまうのである．

次に，第1.2節のプログラムを3回実行したところを示す．

```
% a.out            第1回目の実行
269167349
1169529124
889801541
1254073978
374297457
```

```
2065258565
1482041942
% a.out ──────────── 第2回目の実行
269167349
1169529124
889801541
1254073978
374297457
2065258565
1482041942
% a.out ──────────── 第3回目の実行
269167349
1169529124
889801541
1254073978
374297457
2065258565
1482041942
% _
```

3回とも同じ乱数が発生している．この節では，この状況を改善し，プログラムを実行するたびに異なる乱数が発生するようにする．

2.1◆乱数系列の初期値

乱数の並びを**乱数系列**という．プログラムを実行するたびに同じ乱数系列になる理由は，randの内部処理に秘密がある．randは，前に発生した乱数を覚えていて，それにある種の演算を施して次の乱数を作っている．

```
乱数1 ──────────── プログラム実行開始は同じ値
乱数2 ──────────── 乱数1にある演算を行って作る
乱数3 ──────────── 乱数2にある演算を行って作る
乱数4 ──────────── 乱数3にある演算を行って作る
………
```

このため，乱数系列の初期値が同じなら，発生する乱数の並びは同じになってしまう．

プログラムの実行開始後，初めて作る乱数の初期値は同じである．したがって，プログラムを実行するたびに同じ乱数系列になるのは当然のことなのである．

2.2◆srand

srandという関数を使うと，乱数系列を変えることができる．すなわち，randが覚えている前回の乱数を変更してしまう．

そこで，プログラムの最初のところでsrandを使って，乱数系列の初期値を変更してやる．こうすれば，実行するたびに同じ乱数が発生することを避けることができる．

srandを使うには，stdlib.hをインクルードする．もっとも，randを使うためにstdlib.hをインクルードしているはずだから，同じファイルを二度インクルードする必要はない．

srandの書式は，次のようになっている．

```
void srand(unsigned int seed);
```

引数にunsigned int型のデータを与える．この値が，新しい乱数系列の初期値になる．

```
srand(100);  ―― この場合，100が新しい乱数系列の初期値
```

2.3◆乱数発生プログラム（第2版）

それでは，第1.2節の「乱数発生プログラム」にsrandを加えて，違う乱数が発生するようにする．

【サンプルプログラム】

```
#include <iostream.h>
#include <stdlib.h>

int main()
{
    int r;

    srand(100);  ―― 追加する
    for (int i = 0; i < 7; i++)
        cout << rand() << endl;

    return 0;
}
```

次に実行例を示す.

【実行例】

```
% a.out ─────────────────── 第1回目の実行
894403337
254681679
815315676
1947980633
2131406420
2078022185
1181060739
% a.out ─────────────────── 第2回目の実行
894403337
254681679
815315676
1947980633
2131406420
2078022185
1181060739
% _
```

確かに，第1.2節のときと違う乱数系列が生成されているのがわかる．しかし，やはり実行するたびに同じ乱数になってしまうことに変わりはない．これは，srandに与える引数がいつも100に固定されているからである．

srandに与える引数にでたらめな数を与えることができれば，異なる乱数系列にすることができる．しかし，そこにrandで生成される値を与えてもダメである．これでは堂々巡りになってしまう．

2.4◆time

srandの引数にでたらめな数を与える良い方法がある．time関数を使うのである．この関数は，1970年1月1日からの経過時間を秒単位で返す．したがって，この関数の値をsrandの引数に与えれば，プログラム起動ごとに異なる乱数系列を発生することができる．

関数timeを使用するには，time.hをインクルードする．

```
#include <time.h>
```

関数timeの書式は，次のとおりである．

time_t time(time_t *t);

引数にtime_t型変数へのポインタを与えると，その変数に結果が代入される．もちろん，timeの関数値にも同じ値が得られる．

time_t型変数に値を代入する必要がなければ（今回はそのケースである），引数に0（ゼロ）を指定すればよい．

time(0); ——— 値は経過時間（単位は秒）

したがって，srandの使い方は次のようになる．

srand(time(0));

ただし，これでは型が合わない．

srandの引数の型：unsigned int
timeの型　　　　：time_t

そこで，次のように型キャストする．

srand((unsigned int) time(0));

これで，乱数の初期値を毎回変えることができる．

2.5◆乱数発生プログラム（第3版）

srandの引数にtimeを与え，「乱数発生プログラム」で毎回違う乱数が発生するように改良したものを示す．

【サンプルプログラム】

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h> ——— timeを使うために必要

int main()
{
    int r;

    srand((unsigned int) time(0)); ——— ここを書き換えた
```

```
    for (int i = 0; i < 7; i++)
        cout << rand() << endl;

    return 0;
}
```

次に実行例を示す.

【実行例】

```
% a.out ―――――――――――――――― 第1回目の実行
1163079302
1619643681
2118661563
1931104035
1740483574
129355516
339792449
% a.out ―――――――――――――――― 第2回目の実行
605409212
1914087354
2096217426
723374968
1346088823
172867950
1594873968
% a.out ―――――――――――――――― 第3回目の実行
1055175711
9979304
1422893331
998724974
251664533
1478240955
592613874
% _
```

3 範囲の制御

これまでの実行例を見ると，randはかなり大きな数の乱数を発生しているのがわかる．

```
269167349
1169529124
889801541
1254073978
374297457
2065258565
1482041942
```

これらの乱数は，どのような範囲の数だろう？

- randで発生する最小値はいくつだろう？
- randで発生する最大値はいくつだろう？

このことがわからないと，たとえば1～6までの乱数を作りたいといった応用が利かない．この節では，任意の範囲の乱数を作る方法を説明する．

3.1◆RAND_MAX

まずrandが作る乱数の範囲であるが，これは次のようになっている．

```
0 <= rand() <= RAND_MAX
```

つまり，rand関数は，0以上RAND_MAX以下の整数の乱数を作る．ここで，RAND_MAXは記号定数で，stdlib.hの中で定義されている．次は，GNU Cで提供されているstdlib（/usr/inclue/stdlib.h）の一部である．

```
#define RAND_MAX        2147483647
```

プログラマーは，RAND_MAXがいくつに定義されているかを気にする必要はない．ただ，rand関数が0以上RAND_MAX以下の整数の乱数を作ることだけを知っていれば，自由な範囲の乱数を作ることができる．

3.2◆0以上1未満の乱数

手始めに0以上1未満の乱数を作る．ただし，整数に限定すると，0以上1未満の整数は0しか存在しないので，実数の乱数を作ることにする．これは，次のようにして作ることができる．

注意 数学（といっても簡単な不等式）が苦手な人は，次の部分がわかりづらいかもしれない．その場合は，途中を飛ばして最後の結果だけを見ること．

まず，randが作る乱数が整数であることを念頭に置いて，次の式を見る．

$$0 \leqq \text{rand()} \leqq \text{RAND_MAX}$$

これは，randの定義から事実である．ここで右辺を1つ大きくする．すると，次の式が成立する．

$$0 \leqq \text{rand()} \leqq \text{RAND_MAX} < \text{RAND_MAX} + 1$$

$$\therefore 0 \leqq \text{rand()} < \text{RAND_MAX} + 1$$

つまり，乱数の範囲が「以下」から「未満」に変わったことに注意．

次に，この各辺をRAND_MAX＋1で割る．不等号で結ばれた式は正の数で割っても向きが変わらないので，次の式が成立する．

$$\frac{0}{1 + \text{RAND_MAX}} \leqq \frac{\text{rand()}}{1 + \text{RAND_MAX}} < \frac{1 + \text{RAND_MAX}}{1 + \text{RAND_MAX}}$$

$$\therefore 0 \leqq \frac{\text{rand()}}{1 + \text{RAND_MAX}} < 1$$

この真ん中の式に注目すると，rand()/(1+RAND_MAX)が0以上1未満になっているのがわかる．ただし，これではこの値が整数になってしまうので，0を0.0，1を1.0のように書く．演算の一部に実数が入っていると，結果も実数になるからである．

$$0.0 \leqq \frac{\text{rand()}}{1.0 + \text{RAND_MAX}} < 1.0$$

これで結論が出た．0以上1未満の実数の乱数を作りたければ，rand関数を次のように使えばよい．

```
rand() / (1.0 + RAND_MAX)
```

3.3◆0以上N未満の乱数

0以上1未満の乱数を利用すると，0以上N未満の乱数を作ることができる．作り方は簡単で，単にN倍すればよい．まず，第3.2節の結論は次のようになっていた．

$$0.0 \leqq \frac{\text{rand()}}{1.0 + \text{RAND_MAX}} < 1.0$$

この式を変形していくのであるが，真ん中の部分がごちゃごちゃしているので，次のように記号RANDで置き換える．

$$\text{RAND} = \frac{\text{rand()}}{1.0 + \text{RAND_MAX}} \quad \text{(1)}$$

すると，もとの式を次のように簡単に書くことができる．

$$0.0 \leqq \text{RAND} < 1.0$$

このすべての辺をN倍する（N≧0とする）．N倍しても不等号の向きは変わらないので，次のようになる．

$$0.0 \times N \leqq \text{RAND} \times N < 1.0 \times N$$
$$\therefore 0.0 \leqq \text{RAND} \times N < N$$

これで，RAND × Nが0以上N未満の乱数になることがわかったので，RANDを（1）を使ってもとの値に戻す．すると，次のようになる．

```
(rand() * N) / (1.0 + RAND_MAX)
```

この式を使えば，0以上N未満の乱数を作ることができる．

3.4◆MIN以上MAX以下の乱数

さらに応用として最小値と最大値を指定したMIN以上MAX以下の整数の乱数を作ることもできる．ここで，MAX未満ではなく，MAX以下であることに注意．ま

た，作る乱数が整数であることにも注意．

それでは，その作り方を説明する．まず第3.3節の結論から，次が0以上N未満の乱数である（わかりやすくするため，RANDで置き換えてある）．

$$0.0 \leqq \text{RAND} \times \text{N} < \text{N}$$

この乱数は実数なので，intでキャストして整数に変換する．

$$0 \leqq (\text{int})\ \text{RAND} \times \text{N} < \text{N}$$

未満ではなく以下にするため，右辺の値を1つ小さくする．

$$0 \leqq (\text{int})\ \text{RAND} \times \text{N} \leqq \text{N} - 1$$

このNの代わりにMAX - MIN + 1.0を代入する．

$$0 \leqq (\text{int})\ \text{RAND} \times (\text{MAX} - \text{MIN} + 1) \leqq \text{MAX} - \text{MIN} + 1 - 1$$

これを計算すると，次のようになる．

$$0 \leqq (\text{int})\ \text{RAND} \times (\text{MAX} - \text{MIN} + 1) \leqq \text{MAX} - \text{MIN}$$

この各辺にMINを加える．

$$\text{MIN} \leqq (\text{int})\ \text{RAND} \times (\text{MAX} - \text{MIN} + 1) + \text{MIN} \leqq \text{MAX} - \text{MIN} + \text{MIN}$$

これを計算すると，次のようになる．

$$\text{MIN} \leqq (\text{int})\ \text{RAND} \times (\text{MAX} - \text{MIN} + 1) + \text{MIN} \leqq \text{MAX}$$

これで，真ん中の式でMINがMIN以上MAX未満の乱数になることがわかる．RANDをもとの値に戻すと，次のようになる．

$$(\text{int})\ \frac{\text{rand()}}{1.0 + \text{RAND_MAX}} \times (\text{MAX} - \text{MIN} + 1) + \text{MIN}$$

これをプログラムで書けるように書き換えると，次の式が得られる．

```
MIN + (int) (rand() * (MAX - MIN + 1.0) / (1.0 + RAND_MAX))
```

これが，MIN以上MAX以下の整数の乱数である．

3.5◆サイコロの目

前項で求めた結果を使って，サイコロの目――すなわち，1〜6までの乱数を生成する．この場合，最小値と最大値は次のようになる．

MIN = 1 （最小値）
MAX = 6 （最大値）

これを前項で求めた結果に代入する．

MIN + (int) (rand() * (MAX - MIN + 1.0) / (1.0 + RAND_MAX))
1 + (int) (rand() * (6.0 - 1.0 + 1.0) / (1.0 + RAND_MAX))
1 + (int) (rand() * 6.0 / (1.0 + RAND_MAX))

この式を使うと，1〜6までの乱数を生成することができる．次に，これを使ったサンプルプログラムを示す．

【サンプルプログラム】

これは，1〜6までの乱数を100個作るプログラムである．

```
#include <iostream.h>
#include <stdlib.h>    // for srand(), rand()
#include <time.h>      // for time()

int main()
{
    srand((unsigned int) time(0));
    for (int i = 0; i < 100; i++)
        cout << 1 + (int) (rand() * 6.0 / (1.0 + RAND_MAX))
             << " ";
    cout << endl;

    return 0;
}
```

【実行例】

```
% a.out
5  2  6  5  2  5  5  6  5  3  6  1  6  3  4  6  5  5  5  3  4  2  4  1  1
1  4  5  4  2  4  2  4  3  1  5  2  6  5  1  3  4  2  2  1  5  2  5  4  6
2  1  2  6  1  2  6  5  6  4  6  4  6  4  1  6  2  3  6  6  3  2  4  5  4
4  3  5  2  6  5  4  1  6  3  1  1  2  6  1  6  6  5  5  3  5  5  4  1  4
% _
```

第30章 ファイル入出力：stdio版

この章と次の章で，ディスクファイルに対する入出力を説明する．

第1部の第3章第2.2節で説明したように，もともとC++は入出力機能を持っていない．このため，C++で入出力を行うには，自分で必要なルーチンを書くか，次のいずれかのライブラリを利用する必要がある．

ストリームI/Oライブラリ
stdioライブラリ

この章では，stdioライブラリを使用したファイル入出力を説明する．ストリームI/Oライブラリを利用したファイル入出力については，次の章で説明する．

1 stdioライブラリが必要な理由

ストリームI/Oライブラリは，C++の作者でもあるストロウストラップが作った入出力ライブラリである．これまでも標準入出力を処理するのにストリームI/Oライブラリを使用してきた．ストリームI/Oライブラリを使えば，ディスクファイルへの入出力を行うこともできる．

一方，**stdioライブラリ**は，C++の祖先に当たるC言語の標準入出力ライブラリである．したがって，C++でファイル入出力を行うなら，stdioライブラリを使うよりもストリームI/Oライブラリを使うほうが適している．

では，敢えてこの章でstdioライブラリを取り上げるのはなぜだろう？——それは，C++でも，時にはstdioライブラリが必要になることがあるからである．

プログラムを再利用可能な形でまとめたものを**ライブラリ**という．C++は，いろいろなライブラリが利用できる．

ライブラリは，C言語で書かれたものもあれば，C++言語で書かれたものもある．C言語で書かれたライブラリは，入出力ライブラリとしてstdioライブラリを利用しているものが多い．たとえば第32章で取り上げるcursesライブラリもそうである．

cursesのようなstdioライブラリを前提としたライブラリを使う場合は，入出力ライブラリとしてストリームI/Oを使うわけにはいかない．なぜなら，1つのプログラムの中でstdioライブラリとストリームI/Oライブラリを混在させて使うことは望ましくないからである．

というわけで，いくらC++を使うからといって，stdioライブラリが不要なわけではないのである．

次の節からは，stdioライブラリを使ったファイルの入出力を説明していく．

2 ファイルからの入力

最初に，まず，ファイルからデータを入力する場合について説明する．簡単にその手順だけを説明すると，次のようになる．

ファイルを入力用にオープン

↓

データを入力

↓

ファイルをクローズ

なお，stdioライブラリを使うときは，iostream.hの代わりにstdio.hをインクルードする．

2.1◆ファイルのオープン

ファイルからデータを読み込むには，入力用に，そのファイルをオープンしなければならない．ファイルをオープンするには，標準関数のfopenを使う．次にfopenの書式を示す．

```
FILE *fopen(char *filename, char *mode);
```

これを見るとわかるように，fopenは，**FILE**型のポインタを返す．FILE型とは，ヘッダファイルstdio.hの中で定義されている型である．そこで，まずfopenが返す値を保存するため，FILE型のポインタを用意する．

```
FILE *fp; ―――――― (1)
```

次に，fopen関数を使ってファイルをオープンする．

(1)で用意したポインタ
オープンするファイル名
入力用にオープンすることを示す

```
fp = fopen("test", "r");
```

第1引数には，入力するファイルの名前を指定する．第2引数には**オープンモード**を指定する．次に，オープンモードについて説明する．

2.2◆オープンモード

open関数の第2引数に指定するオープンモードというのは，どのような目的でそのファイルをオープンするのかを示すものである．次のいずれかの記号を指定する．

オープンモード	意味
r	読み込み用にオープンする ファイルが存在しないか，読み込めないときはエラー 読み込み位置はファイルの先頭
r+	読み書き両用にオープンする 読み書きの位置はファイルの先頭
w	書き込み用にオープンする ファイルが存在すれば，その内容をゼロにする
w+	読み書き両用にオープンする ファイルが存在しないときは新規作成する ファイルが存在すれば，その内容をゼロにする
a	内容を追加するためにオープンする ファイルが存在しないときは新規作成する 書き込み位置はファイルの最後

a+	読み書き両用にオープンする ファイルが存在しないときは新規作成する 読み書きの位置はファイルの最後

というわけで，ファイルを読み込み用にオープンするときは，オープンモードに "r" を指定する．

```
fp = fopen("test", "r");
```

これで，ファイルtestはオープンされ，ポインタfpにtestのファイルポインタが保存される．

2.3◆オープン時のエラーチェック

ファイルをオープンするときは，指定したファイルが見つからないなど，ファイルのオープンに失敗することがある．そこで，ファイルオープンしたら，うまくオープンできたかどうかチェックするようにする．

fopenはオープンに失敗するとゼロを返す．そこで，次のようにエラーをチェックすることができる．

```
if ((fp = fopen("test", "r")) == 0)
    // オープンに失敗したときの処理
```

あるいは，次のように書いてもよい．

```
if (!(fp = fopen("test", "r")))
    // オープンに失敗したときの処理
```

2.4◆データの入力

ファイルのオープンに成功したら，入力用の関数を使って，ファイルからデータを入力する．入力用の関数として使いやすいのはfgetcである．

```
c = fgetc(fp);    ――― cに1文字入力
                       ファイルの最後またはエラーだとEOFを返す
```

fgetcは，ファイルから1文字入力する関数である．引数には，fopenから受け取ったファイルポインタを指定する．fgetcが入力に成功すると，入力した文字を返

す．もしファイルの最後に達していたり，入力に失敗すると-1を返す．

fgetcを使ってファイルの全データを入力するには，次のようなループを作ればよい．

```
while ((c = fgetc(fp)) != -1)
    // 入力文字cを処理する
```

2.5◆ファイルのクローズ

データの入力が終了したら，オープンしたファイルを**クローズ**する．ファイルのクローズには，標準関数のfcloseを使う．

fcloseの引数には，オープン時に受け取ったファイルポインタを指定する．fcloseは，クローズに成功すると0を返す．また，失敗すると-1を返す．ファイルを読み込み用に使った場合はエラーチェックは必要ないが，書き込み用に使った場合はエラーチェックをしたほうがよい．ファイルをクローズするとき，バッファに残されたデータがディスクに書き込まれるので，それをチェックするためである．

```
if (fclose(fp))
    // クローズに失敗したときの処理
```

3 メッセージの出力

次に「ファイルからの入力についてのサンプルプログラム」を示すわけであるが，その前にstdioライブラリを使った場合の**メッセージの出力**について説明しておく．なぜなら，stdioライブラリを使うとcoutが使えなくなるからである．メッセージを出力できないと，サンプルプログラムを作るのが厳しくなる．

3.1◆標準出力への出力

stdioライブラリを使って標準出力へメッセージを出力するにはいろいろな関数が利用できるが，ここではprintfを説明する．printfは，さまざまなデータ型を文字列に変換しながら，標準出力に出力する．

printfの最も簡単な使い方は，引数に出力する文字列を指定するものである．た

とえば以下は，標準出力に“Hello, world”の文字列を出力する．

```
printf("Hello, world");
```

3.2◆エスケープシーケンス

printfの第1引数に指定する文字列は，**書式文字列**と呼ばれる特別な文字列である．書式文字列の中には，次の2種類の特別な文字列を含めることができる．

エスケープシーケンス
変換指示子

ここでは，エスケープシーケンスについて説明する．

エスケープシーケンス（*escape sequence*）というのは，文字列中に置かれる特別な文字列で，その部分だけ特別に解釈される．

エスケープシーケンスの先頭の文字は決まっていて，**エスケープ文字**という．printfの書式文字列の中で使われるエスケープ文字はバックスラッシュ（\）である．printfは書式文字列中に文字\を見つけると，そこをエスケープシーケンスの始まりであると解釈する．

以下に，書式文字列で使えるエスケープシーケンスをいくつか示す．

エスケープシーケンス	機能
\a	ベルを鳴らす
\n	改行する
\x	16進数

次にエスケープシーケンスの使用例を示す．

```
printf("Hello, world\n");
```

——“Hello, world”を出力した後，改行する

```
printf("\aHello, world\n");
```

——ベルを鳴らした後“Hello, world”を出力し改行する

これらのエスケープシーケンスは，printfのみならず，coutで出力する文字列の

中でも使用することができる．

> **注意**　エスケープシーケンスは，文字列だけでなく，文字リテラルの中でも使用できる．\xの使い方と合わせて，第32章第7.2節を参照されたい．

3.3◆変換指示子

printfの**書式文字列**に含めることのできる特別な文字列の2つ目は，**変換指示子**である．変換指示子は，10進数とか16進数とかのデータを文字列に変換するときに使用する．

最初に，いくつかの変換指示子を示した後，その使い方を説明する．

変換指示子	代入可能な型	出力
%c	文字	文字
%d	整数	符合付き10進整数
%o	整数	符合なし8進整数
%x	整数	符合なし16進数
%f	浮動小数点数	[-]dddd.dddd
%p	ポインタ	ポインタのアドレス
%s	文字列	文字列

たとえば，次のようなprintfがあるとする．

```
printf("値は%dです\n");
```

この書式文字列には%dという変換指示子が含まれている．%dは整数を文字列に変換する．書式文字列の中に変換指示子を含めたときは，変換するデータを第2引数以下に指定する．たとえば，この例の場合は，int型のデータを指定する．

```
printf("値は%dです\n", 100);
                       ↑
                       ここ
```

この結果，第2引数の100が第1引数の%dに代入され，“値は100です”の文字列が標準出力に出力される．

3.4◆可変引数

複数の変換指示子を指定することもできる．その場合は，変換指示子と同じ数の引数を追加する．

```
printf("値は%dと%fです\n", 100, 2.8);
```

たとえば，この例の場合，第2引数以下の値が次のように代入される．

```
%d ← 100
%f ← 2.8
```

この結果，“値は100と2.8です”という文字列が標準出力に出力される．

このようにprintfは，使用する変換指示子の数に応じて引数の数が変わる．このような関数を**可変引数**をとる関数という．printfは可変引数をとる代表的な関数である．

3.5◆標準エラー出力

標準エラー出力にエラーメッセージを出力するときは，printfの前に“f”を付けたfprintf関数を使う．

```
fprintf(stderr, "エラーメッセージ\n");
```

fprintfは，printfと違って書式文字列の前にstderrを指定する．stderrが何を意味しているかについては第6.2節で説明する．

第2引数は書式文字列である．この中に変換指示子を含める場合は，第3引数以下を追加する．たとえば次は，ファイルのオープンに失敗したときのエラーメッセージを表示するものである．

```
fprintf(stderr, "%s : can't open\n", filename);
```

4 簡易cat

ディスクファイルからデータを入力する例としてcatコマンドの簡易版を作成する．ファイルを入力用にオープンし，そのファイルから入力した文字を標準出力

に出力し，ファイルをクローズして終了するまでの手順が示されている．

【サンプルプログラム】

これは，引数で指定されたファイルの内容を標準出力に出力する簡易catである．ファイルはいくつでも指定することができる．複数のファイルが指定された場合は，それらを順に出力する．1つのファイルにリダイレクトすれば，複数のファイルが連結される．

```
#include <stdio.h>

int main(int ac, char *av[])
{
    int  c;
    FILE *fp;

    for (int i = 1; i < ac; i++) {
        if (!(fp = fopen(av[i], "r"))) {
            fprintf(stderr, "%s :ファイルをオープンできません\n", av[i]);
                return 1;
        }
        while ((c = fgetc(fp)) != -1)
            putchar(c);
    }
    fclose(fp);

    return 0;
}
```

【実行例】

```
% a.out hello ―――――――――― 存在しないファイルを指定
hello :ファイルをオープンできません
% a.out mycat.C ―――――――――― 自身のソースファイルを指定
#include <stdio.h>

int main(int ac, char *av[])
{
    int  c;
    FILE *fp;
    …………
    return 0;
```

```
}
% a.out file1 file2 file3
←———— file1，file2，file3の内容が順に表示される
% a.out file1 file2 file3 > files ———— file1，file2，file3をfilesに連結する
% _
```

プログラムのポイントをいくつか説明する．まず，stdioライブラリを利用するので，iostream.hではなくstdio.hをインクルードする．

```
#include <stdio.h>
```

ファイルをオープンするときエラーになった場合は，第3.5節で説明したfprintfでエラーメッセージを表示している．

```
if (!(fp = fopen(av[i], "r"))) {
    fprintf(stderr, "%s :ファイルをオープンできません\n",
        av[i]);
    return 1;
}
```

このときreturn文でプログラムを終了している．main関数内で使われたreturn文は，プログラムを終了させる．1は終了コードで，エラーが原因で終了したことを示す．

5 ファイルへの出力

次に，データをファイルに出力する方法を説明する．ファイルへの出力も手順は入力のときとほとんど同じである．しかし，書き込みゆえの注意点がいくつか存在する．

5.1◆ファイルのオープン

出力用にファイルをオープンするときも関数fopenを使う．このとき第2引数のオープンモードに "w" を指定する．

```
                     ┌──出力用にオープンすることを示す
                     ▼
fp = fopen("test", "w");
```

出力の場合は入力と違って，指定したファイルが存在しなくてもエラーにはならない．代わりに，指定された名前の新規ファイルを作成する．

しかし，オープン時のエラーチェックを省略してよいというわけではない．指定したファイルが存在し，かつそのファイルの属性が書き込み禁止になっている場合などで，エラーになるからである．

```
if (!(fp = fopen("test", "w"))) {
    // オープンに失敗したときの処理
```

5.2◆データの出力：1文字単位

オープンに成功したら，データをファイルに出力することができる．1文字単位で出力するには，関数fputcを使う．

```
fputc(c, fp); ───── ファイルfpに文字cを出力する
```

データをファイルに出力する場合，書き込みの途中でディスクがいっぱいになるなど，エラーが発生することがある．このため，エラーチェックを行ったほうが安全である．

fputcは，ファイルの終わりになったとき，およびエラー発生時で，同じ値-1を返す．このため，fputcが-1を返しても，ファイルの終わりなのかエラーなのか判定できない．そこで，標準関数のferrorを使う．ferrorは，エラーが発生していないときは0を返す．

```
fputc(c, fp);
if (ferror(fp))
    // 書き込みエラーが発生している
```

5.3◆ファイルのクローズ

クローズのしかたは入力と同じである．ただし，書き込みのときはエラーチェックが必要である．それは，バッファリングという高速化の手法により，ファイルクローズ時にもファイルにデータを出力することがあるからである．このときエラーが発生する可能性があるので，エラーチェックを行う．

if (fclose(fp))
　　// クローズに失敗したときの処理

5.4◆サンプルプログラム

次に，キーボードから入力したデータをファイルに書き込むプログラムを示す．

【サンプルプログラム】

このプログラムは，キーボードから入力したデータを引数で指定したファイルに書き込む．書き込みのときは，いろいろな場面でエラーチェックを行わなければならない．

```
#include <stdio.h>

int main(int ac, char *av[])
{
    int  c;
    FILE *fp;

    if (ac == 1) {
        fprintf(stderr, "usage: writekey filename\n");
        return 1;
    }

    if (!(fp = fopen(av[1], "w"))) {
        fprintf(stderr, "%s :ファイルをオープンできません\n", av[1]);
        return 1;
    }

    while ((c = getchar()) != -1) {
        fputc(c, fp);
        if (ferror(fp)) {
            fprintf(stderr, "%s :書き込みエラー発生\n", av[1]);
            fclose(fp);
            remove(av[1]);
            return 1;
        }
    }

    if (fclose(fp)) {
        fprintf(stderr, "%s :ファイルをクローズできません\n", av[1]);
```

```
        remove(av[1]);
        return 1;
    }

    return 0;
}
```

【実行例】

```
% writekey ─────────────────── ファイル名を指定しない
usage: writekey filename
% writekey bin ─────────────── binは存在するディレクトリ
bin :ファイルをオープンできません
% ls test.data ────────────── test.dataが存在しないことを確認
ls: test.data: No such file or directory
% writekey test.data ──────── test.dataを指定して実行
g++ - GNU project C++ Compile
The C and C++ compilers are integrated
  ◄──────────────────────────── 行頭でCtrl+Dを押す
% ls test.data ────────────── test.dataが作られたか確認
test.data
% cat test.data ───────────── test.dataの内容を見る
g++ - GNU project C++ Compile
The C and C++ compilers are integrated
% _
```

5.5◆ファイルの削除

データを書き込むときにエラーが発生した場合，ファイルのクローズを忘れやすいので注意！　データを書き込むということは，すでにファイルがオープンされているわけなので，クローズしなければならない．

このプログラムでは，さらに書き込み中のファイルを削除している．ファイルの削除にはremove関数を使う．

```
remove(ファイル名);
```

次に，データ書き込み時のエラー処理を示す．

```
while ((c = getchar()) != -1) {
    fputc(c, fp);
```

```
        if (ferror(fp)) {
            fprintf(stderr, "%s :書き込みエラー発生\n", av[1]);
            fclose(fp);
            remove(av[1]);
            return 1;
        }
    }
```

6 fprintf

これまで，関数fprintfをエラーメッセージを表示するのに使ってきた．

```
fprintf(stderr, "%s :ファイルをオープンできません\n",
    av[1]);
```

しかし，そもそも，この関数はファイルにデータを出力するものである．

6.1◆ファイルへの書き込み

fprintfでファイルへの書き込みができるのは，その書式を見ればわかる．

```
int fprintf( FILE *stream, const char *format, …);
```

第1引数がFILEへのポインタになっている．これは，ファイルをオープンするときに使用するfopen関数が返す値と同じ型である．つまり，次のようにすればファイルtestに“Hello, world”という文字列を書き込むことができる．

```
File * fp = fopen("test", "w");
fprintf(fp, "Hello, world");
```

fprintfが優れているのは，書式文字列に変換指示子が使えるので，いろいろな型のデータを文字列に変換して出力することができることである．もちろん，エスケープシーケンスを使うこともできる．たとえば次は，100と2.8をディスクファイルに書き込んだ後，改行コードを書き込む．

```
fprintf(fp, "%d %f\n", 100, 2.8);
```

6.2◆組み込みのファイルポインタ

fprintfの第1引数はFileへのポインタである．ところが，エラーメッセージの出力にfprintfを使ったときは，第1引数にstderrを指定した．

```
fprintf(stderr, "%s :ファイルをオープンできません\n",
    av[1]);
```

このstderrというのは何だろう？

プログラムが実行されるとき，次の3つのファイルが自動的にオープンされる．

標準入力
標準出力
標準エラー出力

stdioライブラリは，これらのファイルに関するファイルポインタを，次の3つのグローバル変数に自動的に割り当てる．

ファイルポインタ	値
stdin	標準入力
stdout	標準出力
stderr	標準エラー出力

つまり，stdin，stdout，stderrの3つは，**組み込みのファイルポインタ**としてプログラムの中で自由に使うことができる．たとえばstderrは標準エラー出力を意味するので，それをfprintfの第1引数に指定すると，標準エラー出力にデータを出力することができる．

stdoutは標準出力を意味するので，次のようにするとprintfと同じになる．

```
fprintf(stdout, "……");
```

なお，fprintfはファイルへの出力を行う関数なので，第1引数にstdinを指定することはできない．

第31章 ファイル入出力：ストリームI/O版

この章では，**ストリームI/Oライブラリ**を使用したファイルの入出力を説明する．第30章で示したサンプルプログラムのストリームI/O版を用意した．両者を比較すれば，両ライブラリの違いを明確にすることができる．

1 ファイルからの入力

ストリームI/Oライブラリを使ってディスクファイルの入出力を行うには，用途に応じて次のクラスを使用する．

クラス	用途
ifstream	ファイルから入力を行う
ofstream	ファイルへ出力を行う
fstream	ファイルへの入出力を行う

これらのクラスを使用するときは，fstream.hをインクルードする．

```
#include <fstream.h>
```

iostream.hについてはfstream.hの中でインクルードされているので，二重にインクルードする必要はない（インクルードしてもかまわないが…）．つまり，fstream.hをインクルードしておけば，ファイルの入出力もできるし，coutやcinを使うこともできる．

次に「ファイルの入出力を行う手順」であるが，これはstdioライブラリのとき

と同じように，次の順序で処理を行う．

ファイルを入力用にオープン
↓
データを入力
↓
ファイルをクローズ

以下，この手順に沿ってifstreamを使ったファイル入力を説明していく．

1.1◆ifstream

それでは，ストリームI/Oを使ったファイルの入力を説明していく．

まず，絶対に必要なのが，ヘッダファイルfstream.hのインクルードである．ストリームI/Oを使ったファイルの入出力では，ファイルの先頭に次の1行を置くのが鉄則である．

```
#include <fstream.h>      // **必須**
```

ファイルを入力する場合に使用するクラスはifstreamである．そこで，ifstream型のオブジェクトを宣言する．オブジェクトの名前は何でもよい．本書では次の変数名を統一して使っていくことにする．

ファイルの入力用：fi
ファイルの出力用：fo

fiは*file input*の略である．if（*input file*）にしなかったのは，if文のifとバッティングするからである．

というわけで，次の変数宣言を行う．

```
ifstream fi;
```

これで，ifstream型のオブジェクトfiが作られる（コンストラクタが自動的に実行される）．あとは，fiのメンバー関数を順序よく呼び出すことで，ファイルの入出力を進めることができる．

1.2◆ファイルのオープン

ファイルをオープンするには，ifstreamクラスのopenメンバー関数を使う．この関数は，ファイル名を引数にとる．たとえば，ファイルtestをオープンするには，次のようにする．

```
fi.open("test");
```

stdioライブラリの場合は，ファイルをオープンするときに，**オープンモード**をいっしょに指定した．しかし，ストリームI/Oのopenは，オープンモードを必要としない．なぜなら，fiがfstreamクラス（入力用）のオブジェクトであることはわかっているので，敢えてオープンモードを指定しなくても入力用にオープンすることがわかるからである．

次にファイルをオープンするだけのサンプルプログラムを示す．

【サンプルプログラム】

これは，ファイルtestをオープンするだけのプログラムである．実行しても何も起こらない．

```
#include <fstream.h>

int main()
{
    ifstream fi;
    fi.open("test");

    return 0;
}
```

注意　コンストラクタの引数にファイル名を指定すれば，openメンバー関数を呼び出さなくてもファイルをオープンすることができる．

```
ifstream fi("test");
```

詳しくは，第1部の第3章第2.3節で紹介した拙著を参照されたい．

1.3◆オープン時のエラーチェック

openメンバー関数を呼び出すと，オープンに成功したかどうかをオブジェクト自身にセットする．

オープンに成功した：fo =自分自身のアドレス
オープンに失敗した：fo = 0

したがって，foの値を見ればオープンに成功したかどうかを判定することができる．

```
if (fi == 0)
    // オープンに失敗したときの処理
```

あるいは，次のように書いてもよい．

```
if (!fi)
    // オープンに失敗したときの処理
```

次に，第1.2節のサンプルプログラムに，オープン時のエラーチェックを組み込んだものを示す．今度はエラーの検出が行われるので，実行時にファイルtestが存在しないと，エラーメッセージが表示される．

【サンプルプログラム】

```
#include <fstream.h>

int main()
{
    ifstream fi;

    fi.open("test");
    if (!fi) {
        cerr << "can't open test" << endl;
        return 1;
    }

    return 0;
}
```

【実行例】

```
% ls test ———————— "test" が存在しないことを確認
ls: test: No such file or directory
% a.out
can't open test ———————— オープンに失敗
% _
```

1.4◆データの入力

ifstreamクラスでは，演算子>>が多重定義されているので，cinと同じように>>でデータを入力することができる．

```
fi >> num; ———————— ファイルfiから変数numにデータを読む
```

しかし，演算子>>を使うと，空白文字が除去されるなどの（時には余計な）サービスが行われる．そこで，ここではifstreamクラスのgetメンバー関数を使う方法を説明する．この関数を使えば，1文字ずつ，すべての文字を入力することができる．

getには，char型の変数を引数に指定する．

```
char c;
fi.get(c);
```

すると，fiから1文字入力され，変数cに保存される．cの値が変更されるのは，getの引数が参照引数になっているからである．

さらにgetは，正しく入力されたかどうかを値として返す．

```
fi.get(c) == 0 ———————— 入力に失敗した
                         または，ファイルの終わりに達している
fi.get(c) != 0 ———————— 入力に成功した
```

ファイルの終わりに達した場合も0を返すので，次のようなループを作ればファイルの内容をすべて読むことができる．

```
while (fi.get(c))
    // 入力した文字cを処理する
```

重要な注意を1つ——．getで入力した文字を受け取る変数cをint型にしてはいけ

ない．char型にしないとエラーになってしまう．

```
int c; ―――――――― これはダメ
fi.get(c);

char c; ―――――――― これが正解
fi.get(c);
```

これは，getの引数の型がcharだからである．stdioライブラリの場合は，getの関数値がintだったのでintを使用した．間違えやすいので注意を！

1.5◆ファイルのクローズ

ファイルをクローズするときは，ifstreamクラスのcloseメンバー関数を使用する．

```
fi.close();
```

次に，ファイルをオープンして，すぐにクローズするサンプルプログラムを示す．

【サンプルプログラム】

これは，ファイルtestをオープンして，すぐにクローズするだけのプログラムである．実行しても何も起こらない．

```
#include <fstream.h>

int main()
{
    ifstream fi;

    fi.open("test");
    fi.close();

    return 0;
}
```

2 簡易cat

第30章第4節で紹介したのと同じ仕様を持つ簡易catのストリームI/O版を作成する．

【サンプルプログラム】

これは，引数で指定されたファイルの内容を標準出力に出力する簡易catである．ファイルはいくつでも指定することができる．複数のファイルが指定された場合は，それらを順に出力する．1つのファイルにリダイレクトすれば，複数のファイルが連結される．

```
#include <fstream.h>

int main(int ac, char *av[])
{
    char c;                                    // int c;はダメ
    ifstream fi;

    // 引数で指定されたファイルごとに
    for (int i = 1; i < ac; i++) {
        // ファイルのオープン
        fi.open(av[i]);
        if (!fi) {
           cerr << "can't open test" << endl;
           return 1;
        }

        // ファイルの終わりまで読む
        while (fi.get(c))
            cout << c;

        // ファイルのクローズ
        fi.close();
    }

    return 0;
}
```

【実行例】

```
% a.out hello ―――――――――――――――――― 存在しないファイルを指定
hello :ファイルをオープンできません
% a.out mycat.C ――――――――――――――――― 自身のソースファイルを指定
#include <fstream.h>

int main(int ac, char *av[])
{
    char c;                                     // int c;はダメ
    ifstream fi;
    ............
    return 0;
}
% a.out file1 file2 file3
  ←―――――――――――――――――――――――― file1, file2, file3の内容が順に表示される
% a.out file1 file2 file3 > files ―――― file1, file2, file3をfilesに連結する
% _
```

3 ファイルへの出力

データをファイルに出力する手順も，入力のときとほとんど同じである．

ファイルを出力用にオープン
↓
データを出力
↓
ファイルをクローズ

ここでは，入力と異なる点を中心に，データを書き込むときの注意点を説明していく．

3.1◆ofstream

まず，使用するクラスはofstreamである．

```
ofstream fo;
```

ファイルのオープンや，そのエラーチェックは入力のときと同じである．

```
fo.open(av[1]);
if (!fo)
    // エラーに失敗したときの処理
```

1文字入力するときは，メンバー関数のgetを使用した．getに対応する1文字書き込み用のメンバー関数はputである．

```
fo.put(c); ———— ファイルfoに文字cを書き込む
```

ファイルをクローズするのも入力と同じである．ただし，書き込みの場合は，エラーのチェックが必要である．この処理は，次のように書くことができる．

```
fo.close(); ———— ファイルをクローズする
if (!fo) ———— クローズに失敗したら
    // エラー処理をする
```

3.2◆サンプルプログラム

第30章第5.4節で示したstdioライブラリによるサンプルプログラムをストリームI/Oライブラリを使って書き換える．このプログラムは，キーボードから入力したデータを，第1引数で指定したファイルに書き込むものである．

【サンプルプログラム】

```
#include <fstream.h>
#include <unistd.h>

int main(int ac, char *av[])
{
    char c;
    ofstream fo;

    if (ac == 1) {
        cerr << "usage: writekey filename" << endl;
        return 1;
    }

    fo.open(av[1]);
```

```
    if (!fo) {
        cerr << av[1]
             << " :ファイルをオープンできません"
             << endl;
        return 1;
    }

    while (cin.get(c)) {
        if (!fo.put(c)) {
            cerr << av[1]
                 << " :書き込みエラー発生"
                 << endl;
            fo.close();
            unlink(av[1]);
            return 1;
        }
    }

    fo.close();
    if (!fo) {
        cerr << av[1]
             << " :ファイルをクローズできません"
             << endl;
        unlink(av[1]);
        return 1;
    }
    return 0;
}
```

3.3◆空白文字を読む

第3.2節のプログラムは，キーボードから1文字入力するのに面白いやり方をしている．これまでなら，次のように書くところであった．

```
cin >> c;
```

しかし，このプログラムは次のように書いている．

```
cin.get(c);
```

getは，ファイルから1文字入力するメンバー関数である（第1.4節参照）．このgetをcinに対しても使うことができる．

```
cin.get(c); ──────── 標準入力から1文字入力し，変数cに保存する
```

演算子の>>を使わなかったのは，>>を使うと空白文字（スペースやタブや改行）が消されてしまうからである．空白文字を含めて入力したいときは，getを使うとよい．

3.4◆ファイルのアンリンク

データの書き込みに失敗したとき，ファイルを削除する．第30章第5.5節では，ファイルを削除するのにremoveを使用した．しかし，この関数はstdioライブラリのものである．stdio.hをインクルードしていないと使えない．

そこで，第3.2節のサンプルプログラムでは，unlinkを使用した．

```
if (!fo.put(c)) {
    cerr << av[1]
        << " :書き込みエラー発生"
        << endl;
    fo.close();
    unlink(av[1]); ◀──────── ここ
    return 1;
}
```

unlinkはOSのシステムコールなので，使用するライブラリに関わりなく，どの環境でも使用することができる．使い方もremoveと同じで，引数に削除するファイルのファイル名を指定する．

unlinkを使用するには，次のファイルをインクルードする必要がある．

```
#include <unistd.h>
```

注意　正しくは，unlinkの機能は指定ファイルのリンク数を1減らすものである．リンク数が0になると，そのファイルは削除される．

第32章 端末プログラミング

stdioライブラリやストリームI/Oライブラリを普通に使って入力ルーチンを書くと，キーボードから入力するときに，最後に必ず Enter キーを押さなければならない．しかし，時には1文字単位で入力したいこともある．

たとえばmoreコマンドは，スペースキーを押した瞬間に1画面スクロールする．スクリーンエディタなどでも，1文字入力するたびにプログラムが反応する．

- 1文字単位で入力する
- カーソルを移動して画面の任意の位置に文字を表示する
- 文字にアンダーラインを付けたり，反転表示したりする
- 色を使う

このようなことを実現するには，システムコールの**ioctl**を使って**端末を制御**する必要がある．しかし，ioctlの使い方はSystemV系とBSD系とで使い方が異なる．そこで，この章では，cursesライブラリを使った端末プログラミングを説明する．

cursesは画面を制御するための高度なライブラリで，ioctlの使い方の違いを吸収してくれる．

なお，cursesには，いくつかのバージョンがある．本書で使用したcursesは，日本語が使えるncurses（*new curses*）である．

1 cursesライブラリのコンパイル

cursesライブラリを使ったプログラムを普通にコンパイルするとエラーになる．

そこで，この節では，cursesライブラリを使ったプログラムをコンパイルするときに注意すべき点を説明する．ポイントは，**ヘッダファイル**と**ライブラリ**である．

1.1◆ヘッダファイル

cursesライブラリを使うには，curses.hをインクルードする必要がある．

```
#include <curses.h>
```

cursesのバージョンによっては，ヘッダファイルの名前が違う名前になっていることがある．たとえばncursesでは，ヘッダファイルの名前がncurses.hになっている（curses.hの場合もある）．その場合でも，ncurses.hがcurses.hにリンクされていることもあるので，ヘッダファイルの名前をよく確認すること．

もう1つは，cursesのヘッダファイルが標準のディレクトリではないところにファイルされている場合である．その場合は，コンパイル時にそのディレクトリを指定する必要がある．使用するオプションは-I（大文字のアイ）である．たとえばcurses.hが/usr/include/ncursesにファイルされているなら，Iオプションを使って次のようにコンパイルする．

```
g++ -I/usr/include/ncurses ファイル名
```

1.2◆ライブラリ

cursesライブラリを使ったプログラムをコンパイルするときに注意しなければならないもう1つの点は，ライブラリの指定である．

cursesは，標準のライブラリではない．したがって，リンクするとき，cursesライブラリもいっしょにリンクするように指定する必要がある．これは，-l（小文字のエル）を使って，次のように指定する．

```
g++ ファイル名 -lcurses
```

ncursesを使う場合は，次のようになる．

```
g++ ファイル名 -lncurses
```

このとき-lとcursesやncursesの間を空けてはいけない．また，-lオプションは，ソースファイル名よりも後ろに指定する．これは，リンクするライブラリの順序に関係がある．cursesライブラリは，いちばん最後に指定する．

なお，1つのシステムに複数のcursesライブラリがインストールされている場合がある．そのときは，使用するヘッダファイルとライブラリを同じにするように注意しなければならない．

2 最初のプログラム

この節から，cursesライブラリを使った具体的なプログラミングに入る．

2.1◆ヘッダファイル

cursesライブラリを使うには，第1節でも説明したようにcurses.hをインクルードする必要がある．次の行をプログラムの先頭に書く．

```
#include <curses.h>
```

注意　1. ncursesを使用する場合，ヘッダファイルの名前がncurses.hになっている場合がある．
2. curses.hをインクルードすると，自動的にstdio.hもインクルードされる．

2.2◆初期化と後処理

cursesライブラリを使用するには，最初に次の関数を実行する必要がある．

```
WINDOW *initscr();
```

この書式を見るとギョッとするかもしれないが，要するに次の1行を書けばよい．

```
initscr();
```

initscrは，最初に1回だけ実行する．2回以上実行してはいけない．initscrの実行により，すべてのcursesデータ構造が初期化される．また，デフォルトのウィンドウが使えるようになる．

cursesライブラリの使用が終わったら，最後に次の関数を実行する．

```
int endwin();
```

これにより，cursesモードが終了する．

2.3◆サンプルプログラム

次にcursesライブラリを使用する最低限のプログラムを示す．

【サンプルプログラム】

このプログラムは，cursesライブラリの使い方を示すものである．initscrを実行後，すぐにendwinを実行している．

```
#include <curses.h>

int main()
{
    initscr();   ―― cursesライブラリを初期化
    endwin();    ―― cursesモードを終了する

    return 0;
}
```

【実行例】

何も起こらない．

2.4◆ベルを鳴らす

前項のサンプルプログラムでは，実行してもすぐに終了してしまうので，cursesライブラリを使っている実感が湧かない．そこで，確かに一度cursesモードに入ったことを確かめるために，次の関数を使ってみる．

```
int beep();
```

この関数はベルを鳴らすためのcursesの関数である．initsrcとendwinの間――すわなち，cursesモードになければ使うことはできない．

```
initscr();
beep();
endwin();
```

【サンプルプログラム】

このプログラムは，beep関数を使ってベルを鳴らすものである．

```
#include <curses.h>

int main()
{
    initscr();  ―― cursesライブラリを初期化
    beep();     ―― ここでベルを鳴らす
    endwin();   ―― cursesモードを終了する

    return 0;
}
```

【実行例】

実行するとベルが鳴って，すぐに終了する．

3 文字の出力

第2.3節では，cursesライブラリを使用する最低限のプログラムを示した．したがって，サンプルプログラムを実行しても何も起こらなかった．

この節では，cursesライブラリで文字を出力する関数を扱う．また，1文字入力する関数も扱う．

cursesライブラリは，画面の出力を最適化している．そのため，リフレッシュという処理を実行しないと，画面には何も表示されない．

3.1◆文字列の出力

cursesライブラリで文字列を出力するには，次の関数を使用する．

```
int addstr(const char *);
```

addstrは，引数に出力する文字列を指定する．たとえば，“Hello, world”という文字列を出力するには，次のようにする．

```
addstr("Hello, world");
```

cursesライブラリを初期化すると，デフォルトで**stdscr**というウィンドウが使えるようになる．stdscrは，端末の画面全体を表すウィンドウである．addstrの出力は，このデフォルトウィンドウに対して行われる．

【サンプルプログラム】

これは，addstr関数を使って“Hello, world”という文字列を出力するものである．ただし，実行しても何も表示されない．

```
#include <curses.h>

int main()
{
    initscr();
    addstr("Hello, world");
    endwin();

    return 0;
}
```

このプログラムを実行しても，すぐに終了してしまう．endwinは，終了時に画面をクリアしてしまうので，“Hello, world”という文字列が出力されたのかどうかわからない．

3.2◆1文字入力

いま示したサンプルプログラムは，実行してもすぐに終了してしまうので，正しく“Hello, world”が表示されたのか確認できない．そこで，endwinを実行する前に1文字入力ルーチンを加え，キー入力があるまでプログラムが終了しないように改良する．

cursesライブラリには**getch**という1文字入力用の関数がある．しかし，これを使うとキーの入力前にrefreshという関数が実行されるので，プログラムによっては画面が乱れることがある．そこで，代わりにstdioライブラリのgetchar関数を使ってもよい．

```
int getchar();
```

ただし，第8節で説明するように，矢印キーなどのファンクションキーを処理したいときは，getchを使う必要がある．

注意
1. refreshについては，第3.3節で説明する．
2. 第2.1節で説明したように，curses.hをインクルードすると自動的にstdio.hもインクルードされる．したがって，getcharを利用するのに，わざわざstdio.hをインクルードする必要はない．

getcharは，入力された文字を返す．したがって，次のようにすれば，入力した文字を変数cに代入することができる．

```
int c = getchar();
```

また，次のように変数に代入しない使い方もある．

```
getchar();
```

この場合は，入力した文字は捨てられてしまう．しかし，キーの入力があるまでプログラムの実行を待つことができる．

【サンプルプログラム】

このプログラムは，前項のサンプルプログラムにgetcharを加え，プログラムがすぐに終了しないようにしたものである．

```
#include <curses.h>

int main()
{
    initscr();
    addstr("Hello, world");
    getchar();          ← 追加した
    endwin();

    return 0;
}
```

【実行例】

実行するとキーの入力待ちになる．何か入力しても何も表示されない．

“Hello, world”の文字列は，表示されなかった．画面に文字列を表示するには，まだ何かが足りないようである．

3.3◆画面出力の最適化

cursesライブラリは，できるだけ画面の出力が高速化されるように最適化されている．すなわち，画面に何かを出力する関数を実行しても，それだけでは何も表示されない．カーソルを移動する関数を実行しても，それだけではカーソルの位置は変わらない．

画面の書き換えが行われるのは，refresh関数が実行されたときである．

```
int refresh();
```

このような仕組みにより，画面の書き換えを最低限で済ませることができる．

```
addstr("文字列1"); ———— これだけでは画面に表示されない
addstr("文字列2"); ———— まだ，画面に表示されない
addstr("文字列3"); ———— まだまだ，画面に表示されない
refresh(); ———————— ここで初めて
                     "文字列1" "文字列2" "文字列3" が
                     表示される
```

というわけで，第3.1節のサンプルプログラムで“Hello, world”を表示させるには，addstrの後でrefresh関数を実行する必要がある．

【サンプルプログラム】

これは，第3.2節のサンプルプログラムにrefresh関数を追加して“Hello, world”の文字列を表示できるように改良したものである．

```
#include <curses.h>

int main()
{
    initscr();
    addstr("Hello, world");
    refresh(); ◀———————— 追加した
    getchar();
    endwin();

    return 0;
}
```

【実行例】

```
Hello, world ―――――――― 画面の左上端に表示される
                        何かキーを押すとプログラムが終了する
```

4 cookedモード

これまで，cursesを使ったいろいろなプログラムを書いてきた．しかし，1文字単位で入力するという所期の目的は，まだ達成されていない．入力の最後に Enter キーを押さなければならなかった．これは，**cookedモード**というUNIXの1モードを使っているからである．

UNIXの端末は，次のようないろいろな**動作モード**で実行される．そして，どのモードにあるかによって端末の動作が変わってくる．

cookedモード
cbreakモード
rawモード
echoモード
crmodモード

各モードは，異なるビットに割り当てられているので，これらのモードを組み合わせて使うことができる．しかし，モードによっては排他的にしか使えないものもある．たとえば，次の3つがそうである．

cookedモード
cbreakモード
rawモード

これらのモードは重要である．これらは，端末によるキー入力のサービスをどの程度行うかを示している．

- **入力は行単位に編集してからプロセスに渡す（バッファリング）**
- **erase処理**
- **kill処理**

これらのモードは排他的なので，どれか1つを選ばなければならない．デフォルトでは，cookedモードになっている．

また，rawモードに設定すると，crmodモードは意味を持たないといった組み合わせ上の注意もある．

この節では，基本となる**cookedモード**における動作を扱う．cookedとは，文字どおり「料理された」モードである．何が料理されるのかというと，入力された文字に対してである．

cookedモードでは，入力した文字に対して，端末が過剰なまでのサービスを行う．たとえば，キー入力された文字はバッファリングされ，Enterキーが押されるまでプログラムに渡されない．

もちろん，cursesライブラリを使用しても，デフォルトの動作モードはcookedモードである．これが，cursesライブラリを使用しても，Enterキーを押さなければならなかった理由である．バッファリングのサービスを停止させるには，第6節のcbreakモードまで待たなければならない．

4.1◆1文字出力

文字列を出力するには，addstr関数を使用した．1文字を出力するには，次の関数を使用する．

```
int addch(const chtype);
```

addchは，引数に出力する文字を指定する．

```
addch('A');  ――― 文字Aを出力する
```

addchもaddstrと同様，refresh関数が実行されるまで画面への表示がお預けになる．

【サンプルプログラム】

このプログラムは，addchによる1文字出力をテストするものである．getcharで入力された文字をaddchで表示している．文字eが入力されると，プログラムが終了する．

```
#include <curses.h>

int main()
{
```

```
    int c;

    initscr();

    addstr("関数addch()のテストです\n");
    addstr("プログラムを終了するには'e'を入力してください\n");
    refresh();

    while ((c = getchar()) != 'e') {
        addch(c);
        refresh();
    }

    endwin();
    return 0;
}
```

このプログラムの主要部分は，次のループ文である．

```
while ((c = getchar()) != 'e') {
    addch(c);
    refresh();
}
```

getcharで1文字入力しては，addchでそれを表示している．ただし，refreshを入れないと画面に表示されないので，addchの後でrefreshを実行している．ループは，文字eが入力されると終了する．

次に，このプログラムの実行例を示す．

【実行例】

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
ABCD ――――― 適当な文字列を入力して[Enter]
ABCD ――――― 入力した文字列が表示される
123456789
123456789
 ◀――――― プログラムを終了するには
           小文字のeを入力して[Enter]を押す
```

> **注意**　cursesのバージョンによっては，この実行例とは違って，入力した文字が表示されないかもしれない．そのようなバージョンについては，第4.3節を参照されたい．

4.2◆バッファリング

前項の実行例を見ると，プログラムとその実行が一致していないことがわかる．すなわち，プログラムの構造は次のようになっていた．

1文字入力
↓
入力した文字を表示する

この繰り返しである．したがって，プログラムどおりに実行されれば，入力した文字がすぐに画面に表示されるはずである．しかし，実際は[Enter]キーが押されるまで入力が続けられ，[Enter]キーが押されて初めて画面への出力が行われた．これが**バッファリング**である．

UNIXの端末は，デフォルトでは**cookedモード**になっている．cookedモードでは，入力された文字がメモリに蓄えられ，すぐにはプログラムに渡らない．

getcharあるいはgetchを実行
↓
キーの入力待ちになる
↓
文字を入力する
↓
入力された文字はバッファに保存される
↓
まだキーの入力待ちが続いている
↓
入力可能なので文字を入力する
↓
入力された文字はバッファに保存される
↓

まだキーの入力待ちが続いている

↓

（Enterキーが入力されるまで続く）

つまり，getchar（あるいはgetch）を実行しても，すぐにはその関数が終了しない．1文字どころか何文字でも入力できる．入力された文字は，どんどんバッファに蓄えられていく．

この状態は，Enterキーが押されて初めて終了する．Enterキーが押されると，バッファに蓄えられている先頭の文字がプログラムに渡され，それがgetcharの関数値になる．プログラムに渡された文字は，バッファから削除される．

前項のプログラムではループ構造になっているので，addchやrefreshが実行され，その文字が表示された後，すぐに2回目のgetcharが実行される．しかし，今度は待たされることはない．すでにバッファには次の文字が入力されているので，その先頭文字がgetcharに渡される．

以下，この繰り返しである．バッファの中の文字がすべてなくなると，再びgetcharで入力待ちになる．そして，Enterが入力されるのを待ち続ける．

これが，前項の実行例で見られた現象の説明である．もし，プログラムで入力された文字をすぐに受け取りたいなら，バッファリングしないようにOSに依頼しなければならない．その方法は第5節以降で説明する．

4.3◆echoモード

ここでは，**echoモード**と**noechoモード**について説明する．これらのモードについては，cookedモードとは直接は関係ない．しかし，cursesのバージョンによる動作の違いを吸収するため，ここでechoモードとnoechoモードの違いについて説明しておく．

第4.1節のサンプルプログラムは，cursesのバージョンによって動作が異なる．第4.1節に示した実行例では，キーを入力するたびに，その文字が画面に表示された．

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
ABCD
```

ABCD ―― 入力した文字が画面に表示される

そして，Enterキーを押すと，同じ文字がもう一度表示された．

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
ABCD
ABCD ―――― 入力した文字列がもう一度表示される
```

それに対して，cursesの別のバージョンでは，キーを入力しても，その文字が画面に表示されない．まるでパスワードを入力しているようである．

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
      ◀―――― ABCDと入力したのだが，
             文字が画面に表示されない
```

そして，Enterキーを押すと，初めて入力した文字が表示される．

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
ABCD ―――― Enterを押すと表示される
```

次に，このバージョンにおける第4.1節の実行例を示す．

【実行例】

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
◀―――― ここで入力待ちになる
   ↓
関数addch()のテストです
プログラムを終了するには'e'を入力してください
◀―――― 適当な文字列を入力する
       (入力した文字は表示されない)
   ↓
関数addch()のテストです
プログラムを終了するには'e'を入力してください
ABCD ◀―――― Enterキーを押すと
            入力した文字列が表示される
   ↓
関数addch()のテストです
プログラムを終了するには'e'を入力してください
```

```
ABCD
123456789
```
← 2行目を入力して[Enter]を押した
eを入力して[Enter]を押すと，
プログラムが終了する

このように第4.1節のサンプルプログラムは，cursesのバージョンによって動作が異なる．その理由は，cursesのデフォルトのモードが，echoモードになっているかnoechoモードになっているかによる．

echoモードとnoechoモードというのは，キーボードから入力した文字を画面に表示するかどうかを示すものである．echoモードは表示するが，noechoモードは表示しない．

echoモード

getchやgetcharで入力した文字を画面に表示する

noechoモード

getchやgetcharで入力した文字を画面に表示しない

推察のとおり，第4.1節で示したcursesは，デフォルトがechoモードになっている．このため，キー入力した文字がすぐに画面に表示された．しかし，ここで示したcursesは，デフォルトがnoechoモードになっている．このため，キー入力した文字は画面に表示されない．

cursesのバージョンによってプログラムの動作が変わってしまうのは困る．そこで，次の関数を使って，echoモード，あるいはnoechoモードに設定できるようになっている．

```
int echo();
int noecho();
```
int echo(); ← echoモードにする
int noecho(); ← noechoモードにする

エディタなどのプログラムは，入力文字をプログラムで処理するので，noechoモードを使う場合が多い．さもないと，たとえば[Ctrl]＋[H]で1文字削除しようとしても“^H”の文字が画面に表示されてしまう．

【サンプルプログラム】

これは，第4.1節のサンプルプログラムをcursesのバージョンにかかわらず，同

じ動作をするように改良したものである．

```
#include <curses.h>

int main()
{
    int c;

    initscr();
    echo();

    addstr("関数addch()のテストです\n");
    addstr("プログラムを終了するには'e'を入力してください\n");
    refresh();

    while ((c = getchar()) != 'e') {
        addch(c);
        refresh();
    }

    endwin();
    return 0;
}
```

このプログラムのポイントは，intscrのすぐ後でecho関数を実行していることである．これによりcursesのバージョンによらずechoモードに設定されるので，どのcursesでも同じように動作する．

【実行例】

第4.1節と同じ．

4.4◆erase処理

cookedモードにおけるサービスの2番目は，**erase処理**である．これは，端末のerase文字に設定されている文字を入力すると，バッファ中の最後の文字を消してくれるものである．どの文字がerase文字に設定されているかは，sttyコマンドで調べることができる．

注意 システムによっては，キーマップの変更をしている場合がある．その場合は，端末の設定と実際のerase文字が異なる場合がある．

第4.3節のプログラムを使って，erase処理を見ることができる．その実行例を次に示す．なお，この実行環境では，Ctrl＋Hがerase文字になっている．

【実行例】

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
123456789^H^H
1234567
```

この実行例では，最初「123456789」を入力した後，Ctrl＋Hを2回入力した．echoモードになっているので，Ctrl＋Hを入力するたびに"^H"が表示される．しかし，同時にcookedモードのerase処理が実行され，バッファから最後の文字（この例では9と8）が削除される．

その後，Enterキーを押すと，バッファの文字がプログラムに渡される．したがって，表示されるのは「1234567」である．

4.5◆kill処理

cookedモードにおけるサービスの3番目は，**kill処理**である．これは，端末のkill文字に設定されている文字を入力すると，バッファ中のすべての文字を消してくれるものである．さらに（可能なら）その行をすべて消してくれる．

どの文字がkill文字に設定されているかは，sttyコマンドで調べることができる．

注意 システムによっては，キーマップの変更をしている場合がある．その場合は，端末の設定と実際のkill文字が異なる場合がある．

第4.3節のプログラムを使って，kill処理を見ることができる．その実行例を次に示す．なお，この実行環境では，Ctrl＋Uがkill文字になっている．

【実行例】

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
1234567 ――――――――――――――― 適当な文字列を入力
  ↓
関数addch()のテストです
プログラムを終了するには'e'を入力してください
  ←――――――――――――――――― Ctrl+Uを押すと消える
  ↓
関数addch()のテストです
プログラムを終了するには'e'を入力してください
ABCDEF ――――――――――――――――― 別の文字列を入力し，Enter
ABCDEF
```

この実行例で，最初に入力した「1234567」がgetcharに渡されていないことに注意．このように，kill処理は，バッファ中のすべての文字を消し去る．

5 改行コードの変換

この節では，**改行コードの変換**について説明する．

cookedモードと第6節で説明するcbreakモードでcrmodモードを選択した場合，Enterキーや Ctrl + M を押すと，ある種の変換が行われる．これが「改行コードの変換」である．この機能は，第7節で説明するrawモードでは使えない．

5.1◆改行コード

UNIXの**改行コード**は10（16進数では0a）である．それを確かめるには，適当なテキストファイルをodコマンドでダンプしてみるとよい．行と行の間に0aが置かれているのがわかる．

この改行コードを端末に出力すると，そこで改行される．また，改行コードを入力すると，バッファリングを終了して，バッファにある文字がプログラムに渡される．

改行コードの0aを入力するには Ctrl + J を押せばよい．試しに，第4.3節のサンプルプログラムを実行してみるとよい．1行入力した後，Ctrl + J を押す．すると，

あたかも[Enter]が押されたように，そこで改行する．

【実行例】

```
関数addch()のテストです
プログラムを終了するには'e'を入力してください
ABCDEFGHIJKL ―――――――――――――― 適当な文字列を入力
  ↓
関数addch()のテストです
プログラムを終了するには'e'を入力してください
ABCDEFGHIJKL
ABCDEFGHIJKL ―――――――――――――― [Ctrl]+[J]を押した
```

まず，ここまでは，UNIXの改行コードが0aであることの説明である．

5.2◆改行コードの変換

続いて，「改行コードの変換」の説明に入る．

まず最初に，次の誤解を解いておく．

「[Enter]キーを押すと改行する」――これは，真っ赤な嘘である．[Enter]キーを押して改行するのは「改行コードの変換」が行われるからである．

では「改行コードの変換」とは何かというと，端末から入力された文字コードの13（16進数では0d）を改行コードの10（16進数では0a）に変換する機能のことである．



0dのコードを入力するには，[Ctrl]+[M]を押せばよい．また，[Enter]キーを押してもよい．[Enter]キーには[Ctrl]+[M]が割り当てられているからである．したがって，どちらのキーを押しても0dを入力することができる．すると「改行コードの変換」機能により，0dが0aに変換される．この結果，改行される．

これが「[Enter]キーを押すと改行する」理由である．

5.3◆nonl

プログラムによっては「改行コードの変換」機能を無効にしたいことがある．cursesの次の関数を使うと，これを無効にすることができる．

```
int nonl();
```

nonl関数を実行するとcrmodモードが解除され，「改行コードの変換」が行われなくなる．逆に，次の関数を使うとcrmodモードに移行し，「改行コードの変換」機能を有効にすることができる．

```
int nl();
```

nlは*new line*，またnonlは*not new line*の略である（と思われる）．

端末は，デフォルトでは「改行コードの変換」機能が有効になっている（crmodモードになっている）．次のプログラムは，nonlを使って「改行コードの変換」機能を無効にしている．このプログラムを実行すると，このサービスの意味を把握することができる．

【サンプルプログラム】

```
#include <curses.h>

int main()
{
    int c;

    initscr();
    echo();
    nonl();    ← 注目！

    addstr("プログラムを終了するには'e'を入力してください\n");
    refresh();

    while ((c = getchar()) != 'e') {
        addch(c);
        refresh();
    }

    endwin();
    return 0;
}
```

このプログラムを実行すると入力待ちになる．適当な文字を入力した後，いつものように[Enter]を押す．しかし，画面には“^M”が表示されるだけである．

【実行例】

```
プログラムを終了するには'e'を入力してください
1234567^M ―――――――――――――――――――――――――― [Enter]を押したが…
```

[Ctrl]＋[M]を押しても，やはり“^M”と表示される．

【実行例】

```
プログラムを終了するには'e'を入力してください
1234567^M^M ――――――――――――――――――――――――― [Ctrl]＋[M]を押しても…
```

[Enter]も[Ctrl]＋[M]も同じコードを入力することがわかる．しかし，nonlを使って「改行コードの変換」機能を無効にしてあるので，ただ“^M”と表示されて改行していない．

改行させるには，[Ctrl]＋[J]を押せばよい．次は，適当な文字列を入力した後，[Ctrl]＋[J]を押したところである．

【実行例】

```
プログラムを終了するには'e'を入力してください
1234567^M^MABC ――――――――――――――――――― ABCを入力してから[Ctrl]＋[J]
ABC
```

今度は，ちゃんと改行して，入力した文字が表示されている．

5.4◆復帰と改行

前項の最後の実行例で[Enter]を入力する前に入力した「1234567」が表示されていないことに注意．

```
1234567^M^MABC ―――――――――― 最後に[Ctrl]＋[J]を押した
ABC ――――――――――――――――――― 「1234567」が消えた
```

これは，Ctrl＋M（画面では“^M”と表示される）とCtrl＋J（画面では“^J”と表示される）が，それぞれ次の意味を持っているからである．

文字コード	キーの入力	名前	機能
0d	Ctrl＋M	復帰	カーソルを行の先頭に戻す
0a	Ctrl＋J	改行	カーソルを1つ下の行に移動する

それでは，Ctrl＋Jが押されてから「ABC」が表示されるまでの動きを見てみよう．

1. ユーザーが「1234567^M^MABC」を押した後Ctrl＋Jを押す．
2. Ctrl＋Jは改行なので，カーソルが1行下に移動する．
   （具体的には文字Cの右下に移動）
3. バッファの先頭から，まず「1234567」が画面に表示される．
   （次のような表示になる）

```
1234567^M^MABC
                1234567
```

4. “^M”でカーソルが行の先頭に戻る．
   （^Mが2回出力されるが同じこと）
5. そこから文字の「ABC」が出力される．
   （次のような表示になる）

```
1234567^M^MABC
ABC            1234567
```

6. 最後に，バッファの最後の文字Ctrl＋Jが出力される．
   このコードをaddchで出力すると，カーソルから右を消した後，改行する．
   （次のような表示になる）

```
1234567^M^MABC
ABC
```

これが，Ctrl＋Jが押されたとき「ABC」だけが表示された理由である．

次にCtrl＋J（画面では“^J”と表示される）の機能を見るための実行例を示す．

【実行例】

```
プログラムを終了するには'e'を入力してください
1234567^M^MABC
ABC
AAA ――――――――――――――――――――― 最後にCtrl+Jを押す
    AAA
BBBB ―――――――――――――――――――― 最後にCtrl+Jを押す
     BBBB
CCCCC ――――――――――――――――――― 最後にCtrl+Jを押す
      CCCCC
```

Ctrl+Jを押すたびにカーソルが1文字下に移動しているのがわかる．

6 cbreakモード

デフォルトのcookedモードを使っているかぎり，あまりcursesを使う有難味が感じられない．cookedモードではバッファリングされるので，1文字単位で入力することはできない．バッファリングをしないようにするには，cbreakモードを使う．

6.1◆モードの設定

cbreakモードに入るには，次の関数を使う．

```
int cbreak();
```

また，cbreakモードを解除するには，次の関数を使う．

```
int nocbreak();
```

cbreakモードを使うときは，併せてnonl（第5節参照）を設定するとよい．nlモードになっているとEnterキーで改行が効くので，エディタなどの処理に向かない．

次に，cbreakモードがどのようなものかを示すためのサンプルプログラムを掲げる．

【サンプルプログラム】

このプログラムは，cursesでcbreakモードを使うものである．getcharで1文字入

力しては，入力した文字をaddchで出力するという処理を，文字のeが入力されるまで繰り返す．

なお，併せて次のモードも設定している．

echo(); ———— 入力した文字をその場で表示する
nonl(); ———— 改行コードの変換をしない

```
#include <curses.h>

int main()
{
    int c;

    initscr();
    echo();
    nonl();
    cbreak();

    addstr("プログラムを終了するには'e'を入力してください\n");
    refresh();

    while ((c = getchar()) != 'e') {
        addch(c);
        refresh();
    }

    endwin();
    return 0;
}
```

【実行例】

```
プログラムを終了するには'e'を入力してください
aabbccdd
```

cbreakモードでは，バッファリングされない．このため，1文字入力するたびにすぐに反応する．eを押せば，即座にプログラムが終了する．

6.2◆割り込みキー

cbreakモードを使うとスクリーンエディタなどを作れるように見えるが，少しまずい点がある．次の割り込みキーやその他のコントロールキーが有効になってしまうからである．

コントロールキー	sttyでの表示	機能
Ctrl＋C	intr	プログラムを終了する
Ctrl＋Z	susp	プログラムを一時停止する
Ctrl＋S	stop	画面への表示を停止する
Ctrl＋Q	rprnt	画面への表示を再開する

※これらのキーのうちCtrl＋Cを割り込みキーという

たとえば，前項のサンプルプログラムを実行中にCtrl＋Cを押すと，プログラムが終了してしまう．それはそれでかまわないのだが，エディタでCtrl＋Cに何か意味を持たせようとしたとすると，プログラムでCtrl＋Cを受け取ることができない（プログラムがCtrl＋Cを受け取る前に，OSがプロセスを終了してしまう）．

Ctrl＋Cを**割り込みキー**という．そもそもcbreakモードのcbreakは，Ctrl＋Cでbreakすることから来ている．

6.3◆一時停止

第6.1節のサンプルプログラムを実行中にCtrl＋Zを押すと，次のメッセージを表示してプログラムが停止してしまう．

```
                    ←―――――――――――― ここでCtrl＋Zを押した
[1]+  Stopped (tty output)   a.out
 % _
```

あたかもプログラムが終了したかに見える．しかし，Ctrl＋Zの場合は一時停止である．実際，psコマンドを実行すれば，それを確認できる．

```
% ps
   PID TTY STAT   TIME COMMAND
   277    1 S      0:00 /bin/login -- kt
   ........................
```

```
    6029  p0 T    0:00 a.out ──────── Tは一時停止を示す
    6030  p0 R    0:00 ps
  % _
```

プログラムを再開するには，**fg**コマンドを使えばよい．

注意　プログラムを再開するコマンドは，シェルによって異なる．

6.4◆表示の停止と再開

第6.1節のサンプルプログラムを実行中にCtrl＋Sを押すと，キーを入力しても何も表示されなくなってしまう．まるで，OSが暴走したかのように見えてしまう．

【実行例】

```
プログラムを終了するには'e'を入力してください
AABBCC ──────── 「ABC」を入力した後「Ctrl+S, 123」と入力した
```

Ctrl＋Sを押した後に入力した「123」が表示されていない．しかし，実際には，ちゃんとキーは入力されている．ただ，画面への表示が一時停止されているだけである．画面への表示を再開するには，Ctrl＋Sを押せばよい．それまで入力したが画面には表示されていなかった文字が，いっきょに画面に表示される．

【実行例】

```
プログラムを終了するには'e'を入力してください
AABBCC123 ──────── Ctrl+Qで「123」が表示される
```

以上見てきたように，cbreakモードではCtrl＋Cの割り込みキーをはじめとするCtrl＋Z，Ctrl＋S，Ctrl＋Qがプログラムには渡されず，特殊な働きをしてしまう．これらのキーをプログラムで処理したいなら，次の節で説明するrawモードを使う必要がある．

7 rawモード

cbreakモードを使うことにより，**バッファリングサービス**を停止させることができる．しかし，依然として[Ctrl]＋[C]の割り込みキーをはじめとする[Ctrl]＋[Z]，[Ctrl]＋[S]，[Ctrl]＋[Q]がプログラムには渡されず，特殊な働きをしてしまう．

端末によるすべてのサービスを停止するには，これから説明する**rawモード**を使う必要がある．

7.1◆モードの設定

rawモードに入るには，次の関数を使う．

```
int raw();
```

また，rawモードを解除するには，次の関数を使う．

```
int noraw();
```

rawモードでは，nl，nonlの設定にかかわらず「改行コードの変換」は行われないので，nonl（第5節参照）の設定は不要である．

次にrawモードがどのようなものかを示すためのサンプルプログラムを掲げる．

【サンプルプログラム】

このプログラムは，cursesでrawモードを使うものである．getcharで1文字入力しては，入力した文字をaddchで出力するという処理を，文字のeが入力されるまで繰り返す．

```
#include <curses.h>

int main()
{
    int c;

    initscr();
    echo();
    raw();
```

```
    addstr("プログラムを終了するには'e'を入力してください\n");
    refresh();

    while ((c = getchar()) != 'e') {
        addch(c);
        refresh();
    }

    endwin();
    return 0;
}
```

【実行例】

```
プログラムを終了するには'e'を入力してください
112233^C^C^Z^Z^R^R^Q^Q
```

もはや[Ctrl]＋[C]，[Ctrl]＋[Z]では，プログラムが終了しない．また，[Ctrl]＋[S]，[Ctrl]＋[Q]も効かないのがわかる．すべてのキーをプログラムで処理することが可能である．

逆にいうと，rawモードは**危険**なモードでもある．プログラムを終了する手段を用意しておかないと（上のサンプルプログラムでは[e]キーで終了できるようにしてある），killコマンドのお世話にならざるを得なくなる．

開発はcbreakモードで始め，終了手段の目途が立ったところでrawモードに移行するとよい．

7.2◆キーコードを調べるプログラム

ここでは，rawモードを使って自分のシステムが使用している**キーコード**を調べるプログラムを示す．これまでとは違って，プログラムを終了するのに[Ctrl]＋[X]を使っているので注意．

【サンプルプログラム】

これは，自分のシステムが使用しているキーコードを調べるためのプログラムである．キーを押すと，そのキーのキーコードが16進数で表示される．

```
#include <curses.h>

int main()
{
    int c;

    initscr();
    noecho();                          ―― キー入力時の表示を停止
    raw();

    addstr("キーコードを調べます\n");
    addstr("プログラムを終了するにはCtrl-Xを入力してください\n");
    refresh();

    while ((c = getchar()) != '\x18') {
        printf("%02x : %c\r\n", c, c);   ―― 改行は\r\n
        refresh();
    }

    endwin();
    return 0;
}
```

【実行例】

```
キーコードを調べます
プログラムを終了するにはCtrl-Xを入力してください
41 : A      ―― 大文字のコード
42 : B
43 : C
61 : a      ―― 小文字のコード
62 : b
63 : c
31 : 1
32 : 2
33 : 3
20 :        ―― これはスペース
01 :        ―― Ctrl＋A
02 :        ―― Ctrl＋B
03 :        ―― Ctrl＋C
```

このプログラムを見れば，たとえばCtrl＋Aのキーコードが01であることがわかる．

プログラムのポイントを説明する．まず，入力したキーを表示しないため，noechoモードに設定している．

```
noecho();
```

プログラムを終了するキーは[Ctrl]+[X]である．そのチェックを行っているのが次のところである．

```
while ((c = getchar()) != '\x18')
```

'\x18'は[Ctrl]+[X]を表す文字リテラルである．文字リテラル''の中には，文字列の中に書くのと同じエスケープシーケンスを使うことができる．[Ctrl]+[X]のキーコードが16進数の18であることは，このプログラムを少し改良すればわかる（このプログラムの場合，[Ctrl]+[X]を押すと，瞬時にプログラムが終了してしまう）．

結果を表示するところは，次のprintfで行っている．

```
printf("%02x : %c\r\n", c, c);
```

%02xは，変換指示子で2桁の16進数で表示することを指定している．

rawモードでは，\nだけでは改行しないので（1文字下に下がるだけなので），\aの復帰コードも指定する必要がある．

8 ファンクションキーの利用

この節では，矢印キーとか[F1]，[F2]，…キーとか[PgUp]，[PgDn]キーといった**ファンクションキー**をプログラムの中で扱う方法を説明する．

8.1◆ファンクションキーのキーコード

まず，第7.2節で示した「キーコードを調べるプログラム」で，ファンクションキーを押したとき，どのようなキーコードが生成されるかを調べる．

次に，その実行例を示す．これは，[↑][↓][←][→]の各キーを押してみたところである．

【実行例】

```
キーコードを調べます
プログラムを終了するにはCtrl-Xを入力してください
1b :  ――――――――――――――――――――― ↑キーを押した
5b : [
41 : A
1b :  ――――――――――――――――――――― ↓キーを押した
5b : [
42 : B
1b :  ――――――――――――――――――――― ←キーを押した
5b : [
44 : D
1b :  ――――――――――――――――――――― →キーを押した
5b : [
43 : C
```

この実行結果を見ると，ファンクションキーを押すと，1bで始まる**複数のキーコード**が発生することがわかる．たとえば，↑キーのキーコードは1b，5b，41の3つである．したがって，少々面倒ではあるが，

キーから入力されたコードが1b，5b，41であれば
↑キーが押された

といったプログラムを書けば，↑キーを処理することができる．

しかし，残念ながら，この方法には汎用性がない．

一般にファンクションキーには，自由なコードを割り当てることができる．上の場合は，たまたま↑キーに1b，5b，41のコードが割り当てられていたが，いつもいつもこのようになっているとは限らない．したがって，ファンクションキーのキーコードを調べて，それでファンクションキーを処理しようとしても，汎用的なプログラムにはならないのである．

8.2◆keypad

cursesライブラリを使えば，ファンクションキーを統一的に処理することができる．それには，次の関数を使用する．

int keypad(WINDOW *, bool);

第1引数は，どのウィンドウを対象とするかを指定する．といっても，本書ではデフォルトウィンドウのstdscrしか扱わないので，stdscrを指定する．

```
keypad(stdscr, …);
```

第2引数にはbool型の値を指定する．bool型はcurses.hの中で定義されていて，次のいずれかの値をとることができる．

```
TRUE
FALSE
```

で，ファンクションキーを処理するときはTRUEを指定する．

```
keypad(stdscr, TRUE);
```

このようにkeypadを実行する．そして，cursesライブラリのgetch関数で，キーコードを取得する（getcharではダメである）．すると，ファンクションキーのキーコードが統一されたものになる．しかも，それらのキーコードには統一された記号定数が定義されている．たとえば，[↑]キーのキーコードはKEY_UPである．したがって，次のようなコードを書けば，[↑]キーを処理することができる．

```
int c = getch();
if (c == KEY_UP)
      // [↑]キーが押されたときの処理
```

どのような記号定数が定義されているかは，curses.hを見ればわかる．次に代表的なファンクションキーの記号定数を掲げる．

ファンクションキー	記号定数
[↑]	KEY_UP
[↓]	KEY_DOWN
[←]	KEY_LEFT
[→]	KEY_RIGHT
[PgUp]	KEY_PPAGE
[PgDn]	KEY_NPAGE
[F1]	KEY_F(1)
[F2]	KEY_F(2)
[F3]	KEY_F(3)
…	……

8.3◆サンプルプログラム

keypad関数を使ってファンクションキーを処理するプログラムを次に示す.

【サンプルプログラム】

このプログラムは，ファンクションキーの入力を検出する．検出するキーは，次の3種類である．

矢印キー
PgUp，PgDn
F1，F2

その他のキーが押された場合は，そのまま表示する．

```
#include <curses.h>

int main()
{
    int c;

    initscr();
    noecho();
    raw();
    keypad(stdscr, TRUE);  ――― 必須

    addstr("ファンクションキーを検出します\n");
    addstr("プログラムを終了するにはCtrl-Xを入力してください\n");

    while ((c = getch()) != '\x18')  ――― getchを使う
        switch (c) {
        case KEY_UP:
            addstr("[↑]");
            break;
        case KEY_DOWN:
            addstr("[↓]");
            break;
        case KEY_LEFT:
            addstr("[←]");
            break;
        case KEY_RIGHT:
            addstr("[→]");
            break;
```

```
        case KEY_PPAGE:
            addstr("[PgUp]");
            break;
        case KEY_NPAGE:
            addstr("[PgDn]");
            break;
        case KEY_F(1):
            addstr("[F1]");
            break;
        case KEY_F(2):
            addstr("[F2]");
            break;
        default:
            addch(c);
        }

    endwin();
    return 0;
}
```

【実行例】

```
ファンクションキーを検出します
プログラムを終了するにはCtrl-Xを入力してください
[↑][←][→][↓][PgUp][PgDn][F1][F2]ABCDEFG
```

押されたキーがファンクションキーなら，[↑] のように，その種類を表示する．また，AとかBとかCなどのように普通のキーなら，そのまま表示する．

9 カーソル移動

スクリーンエディタなどを作成する場合は，カーソルを1文字上に移動するなど**カーソル移動**の処理が必要である．cursesライブラリを使ってカーソルを移動させるには，moveを使う．

move(y座標, x座標);

逆に，現在のカーソル座標を取得するにはgetsyxを使う．

getsyx(y, x);

これで，変数y, xにy座標, x座標が得られる．

座標の数え方は，画面の左上が（0, 0）である．initscrで初期化直後の座標は（0, 0）になっている．moveでカーソルを移動しても，それを画面に反映させるためには，やはりrefreshが必要である．

【サンプルプログラム】

これは，moveとgetsyxの使い方を示すもので，画面に三角形を描く．

```
#include <curses.h>

int main()
{
    int x, y;

    initscr();

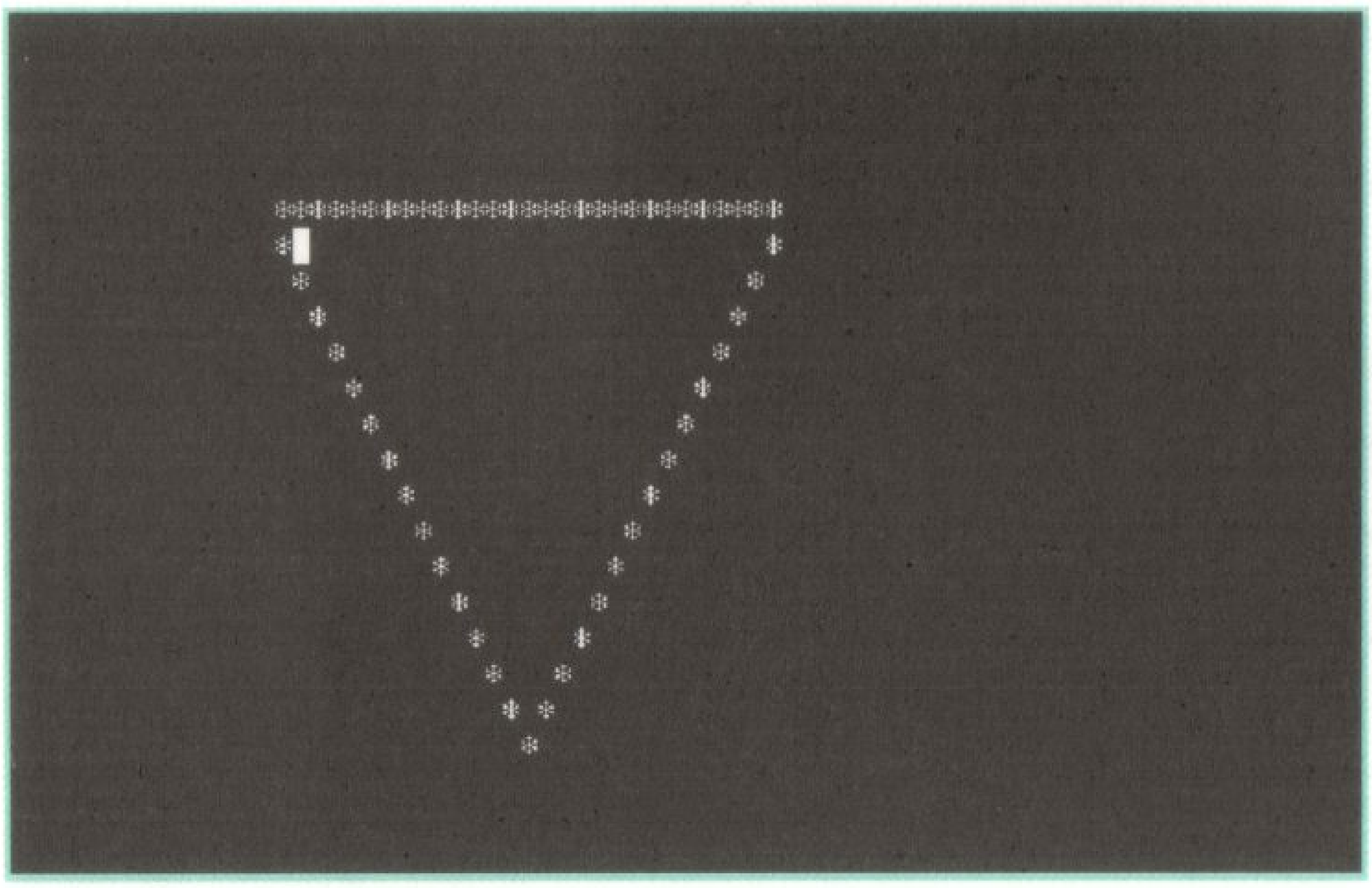
    move(10, 15);
    addstr("******************************");

    getsyx(y, x);
    for (int i = 0; i < 15; i++) {
        move(++y, --x);
        addch('*');
    }
    for (int i = 0; i < 14; i++) {
        move(--y, --x);
        addch('*');
    }

    refresh();
    getchar();
    endwin();
    return 0;
}
```

【実行例】

図32-1▶



プログラムのポイントを説明する．まず，次は三角形の横線を引いている．

```
move(10, 15);
addstr("*****************************");
```

addstrを実行すると，カーソルは最後に描いた文字の右隣に移動する．その座標を次のgetsyxで変数y, xに取得している．

```
getsyx(y, x);
```

次のfor文は，三角形の左下がりの辺を描いている．

```
for (int i = 0; i < 15; i++) {
    move(++y, --x);
    addch('*');
}
```

1回ループするたびにy座標は+1（1行下がる），x座標は-1（1文字左に移動）していることに注意．同様に，次のループで左上がりの辺を描いている．

```
for (int i = 0; i < 14; i++) {
    move(--y, --x);
    addch('*');
}
```

最後にrefreshを実行して表示している．refreshを実行しているのは，ここ1回だけである．addstrやmoveやaddchを実行するたびにrefreshする必要はない．図形

が出来上がった後，まとめて1回実行すれば十分である．

最後のgetchar()は，キー入力があるまで表示を消すのを待つために入れている．これがないと，瞬間的にプログラムが終了してしまうので，せっかく描いた三角形を見ることができない．

10 文字を装飾する

cursesライブラリを利用して，文字にアンダーラインを引いたり，反転表示したりすることができる．使用する関数はattrsetである．

```
int attrset(int attrs);
```

引数には，次のアトリビュートを指定する．

アトリビュート	効果
A_REVERSE	白黒を反転する
A_UNDERLINE	アンダーラインを付加する
A_BLINK	点滅する
A_BOLD	太字で表示する
A_NORMAL	標準に戻す

たとえば，文字を太字で表示するなら，次のようにする．

```
attrset(A_BOLD);
```

この後，文字を表示すると，指定したアトリビュート付きで表示される．もちろん，使用している端末が，そのアトリビュートをサポートしていなければならない．

【サンプルプログラム】

これは，文字の修飾をテストするためのサンプルプログラムである．実行すると，いろいろなアトリビュート付きで文字が表示される．

```
#include <curses.h>

int main()
{
```

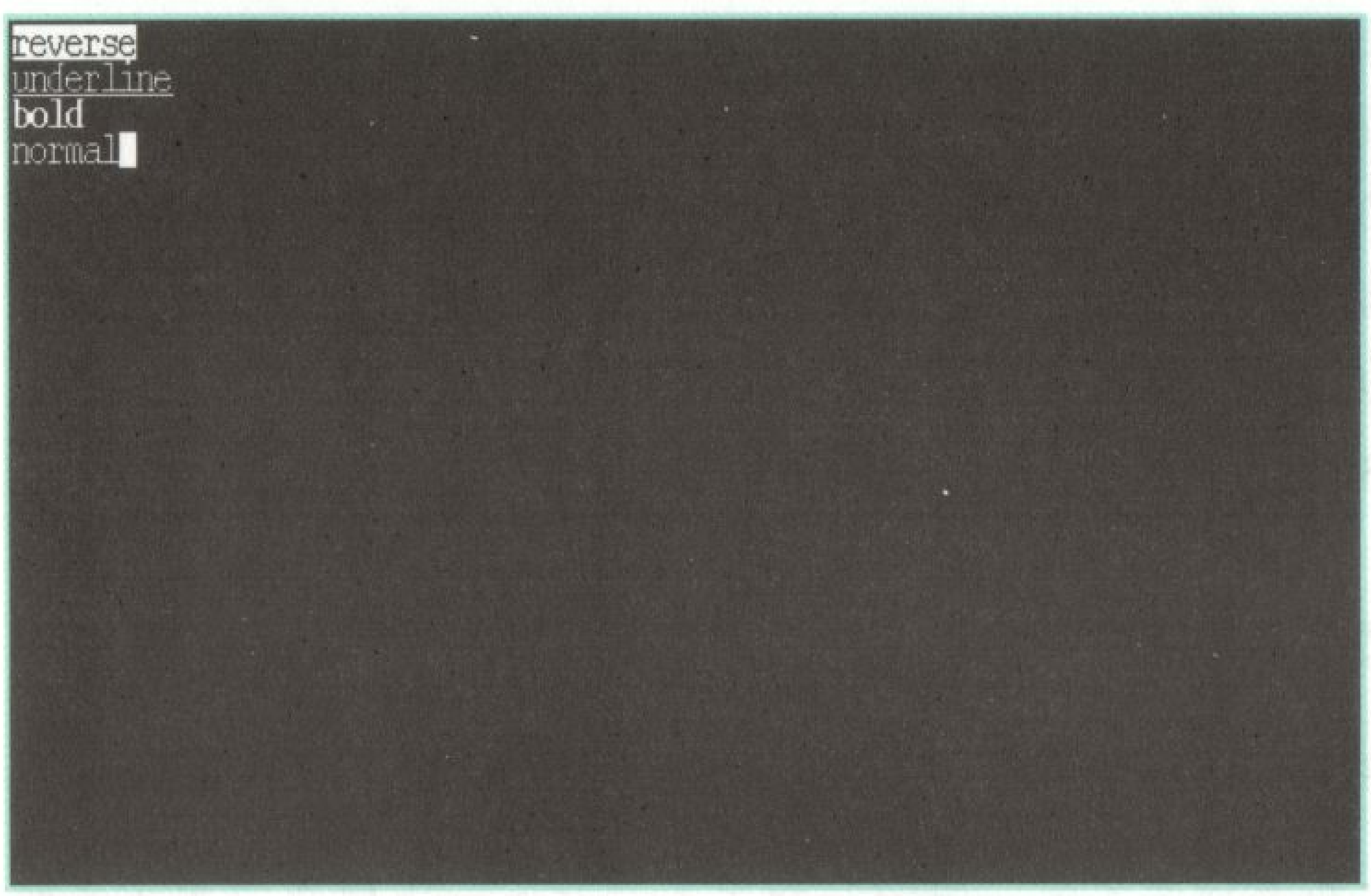
```
    initscr();
    raw();

    attrset(A_REVERSE);
    addstr("reverse\n");
    attrset(A_UNDERLINE);
    addstr("underline\n");
    attrset(A_BOLD);
    addstr("bold\n");
    attrset(A_NORMAL);
    addstr("normal");

    refresh();
    getchar();
    endwin();
    return 0;
}
```

【実行例】

図32-2▶



11 色を付ける

この節では，cursesで「色」を使う方法を説明する．色の設定には，文字の修飾と同じattrsetを使う．

int attrset(int attrs);

文字の修飾ではattrsetの引数にアトリビュートを指定したが，色を付ける場合は，設定する色を指定する．

```
attrset(色の指定);
```

ただし，色の場合は，文字の修飾ほど簡単ではない．色を指定するのに，いろいろな前準備が必要だからである．

11.1◆色を使う準備

色を使用するときは，最初に次の関数を実行しておく必要がある．

```
int start_color();
```

次に，使用する色を定義する．「色を定義する」といってもピンとこないかもしれない．

「赤を使いたければ“赤”と指定すればいいではないか！」

――確かにそうなのであるが，そう単純にはいかない．cursesは，前景色と背景色を別々に指定することができる．

前景色 ―――――――― 文字の色
背景色 ―――――――― 文字の背景の色

つまり，同じ赤でも文字の色が赤なのか，それとも背景の色が赤なのか，また，文字の色を赤にするにしても，バックを黄色にするとか，バックを白にするとか，それこそ自由に設定することができるのである．

そこで，プログラムで使用する前景色と背景色の組み合わせをあらかじめ定義しておき，色を指定するときは，その組み合わせで指定するのである．

11.2◆カラーペア

前景色と背景色の組み合わせを**カラーペア**という．カラーペアを定義するには，次の関数を使用する．

```
int init_pair(short pair, short f, short b);
```

init_pareの第2引数には前景色，第3引数には背景色を指定する．このとき色を指定するのに，次の記号定数を使うことができる．

記号定数	色
COLOR_BLACK	黒
COLOR_BLUE	青
COLOR_RED	赤
COLOR_MAGENTA	深紅色
COLOR_GREEN	緑
COLOR_CYAN	水色
COLOR_YELLOW	黄色
COLOR_WHITE	白

たとえば，文字の色を赤，文字の背景を黄色にするなら，次のように指定する．

```
init_pair(…, COLOR_RED, COLOR_YELLOW);
```

定義したカラーペアは，番号で参照する．その番号をinit_pairの第1引数で指定する．たとえば，いまの赤と黄色のカラーペアを1番に定義するなら次のようにする．

```
init_pair(1, COLOR_RED, COLOR_YELLOW);
```

これで，カラーペアの1番は，赤と黄色の組み合わせということになる．

カラーペアの0番は予約されていて，デフォルトの前景色でデフォルトの背景色，具体的には黒字に白のカラーペアが0番になる．init_pairの定義は，そのデフォルトの定義を除く1番から順に定義していく．番号の最大値は，COLOR_PAIRS - 1である．

COLOR_PAIRSは使用可能なカラーペアの数を表すもので，start_color（色を使うとき最初に実行する関数）を実行すると自動的に設定される．

11.3◆色の設定

カラーペアを定義すると，文字の修飾でも使用したattrsetを使って色を設定することができる．このとき，attrsetには，定義したカラーペアの番号を指定する．

```
attrset(カラーペアの番号);
```

ただし，カラーペアの番号を直接指定することはできない．COLOR_PAIRというマクロ（本書では説明していないが，一種の関数）を仲介に指定する．

```
attrset(COLOR_PAIR(カラーペアの番号));
```

たとえば，第11.2節で定義したカラーペアの1番（文字の色は赤，文字の背景は黄色）を指定するなら次のようにする．

```
attrset(COLOR_PAIR(1));
```

これで，以後addstr関数などで文字を表示すると，文字の色は赤，文字の背景は黄色で表示されるようになる．

なお，カラーペアといっしょにアトリビュートを指定することもできる．たとえば次は，カラーペアの1番とアンダーラインを同時に指定したものである．

```
attrset(COLOR_PAIR(1) | A_UNDERLINE);
```

これにより，文字の色は赤，文字の背景は黄色でアンダーライン付きに設定される．また次は，文字の色は赤，文字の背景は黄色，アンダーライン付きで太字に設定される．

```
attrset(COLOR_PAIR(1) | A_UNDERLINE | A_BOLD);
```

11.4◆サンプルプログラム

色の使い方を示すサンプルプログラムを次に示す．

【サンプルプログラム】

このプログラムを実行すると，いろいろな色で文字が表示される．

```
#include <curses.h>

#define BLACK       0
#define BLUE        1
#define RED         2
#define PURPLE      3
#define GREEN       4
#define CYAN        5
#define YELLOW      6
#define WHITE       7

int main()
{
    initscr();
    start_color();
```

```
    init_pair(BLUE,   COLOR_BLUE,    COLOR_BLACK);
    init_pair(RED,    COLOR_RED,     COLOR_BLACK);
    init_pair(PURPLE, COLOR_MAGENTA, COLOR_BLACK);
    init_pair(GREEN,  COLOR_GREEN,   COLOR_BLACK);
    init_pair(CYAN,   COLOR_CYAN,    COLOR_BLACK);
    init_pair(YELLOW, COLOR_YELLOW,  COLOR_BLACK);
    init_pair(WHITE,  COLOR_WHITE,   COLOR_BLACK);

    move(5, 10);
    attrset(COLOR_PAIR(BLUE));
    addstr("青で表示しています");

    move(6, 10);
    attrset(COLOR_PAIR(RED));
    addstr("赤で表示しています");

    move(7, 10);
    attrset(COLOR_PAIR(PURPLE));
    addstr("紫で表示しています");

    move(8, 10);
    attrset(COLOR_PAIR(GREEN));
    addstr("緑で表示しています");

    move(9, 10);
    attrset(COLOR_PAIR(CYAN));
    addstr("水色で表示しています");

    move(10, 10);
    attrset(COLOR_PAIR(YELLOW));
    addstr("黄色で表示しています");

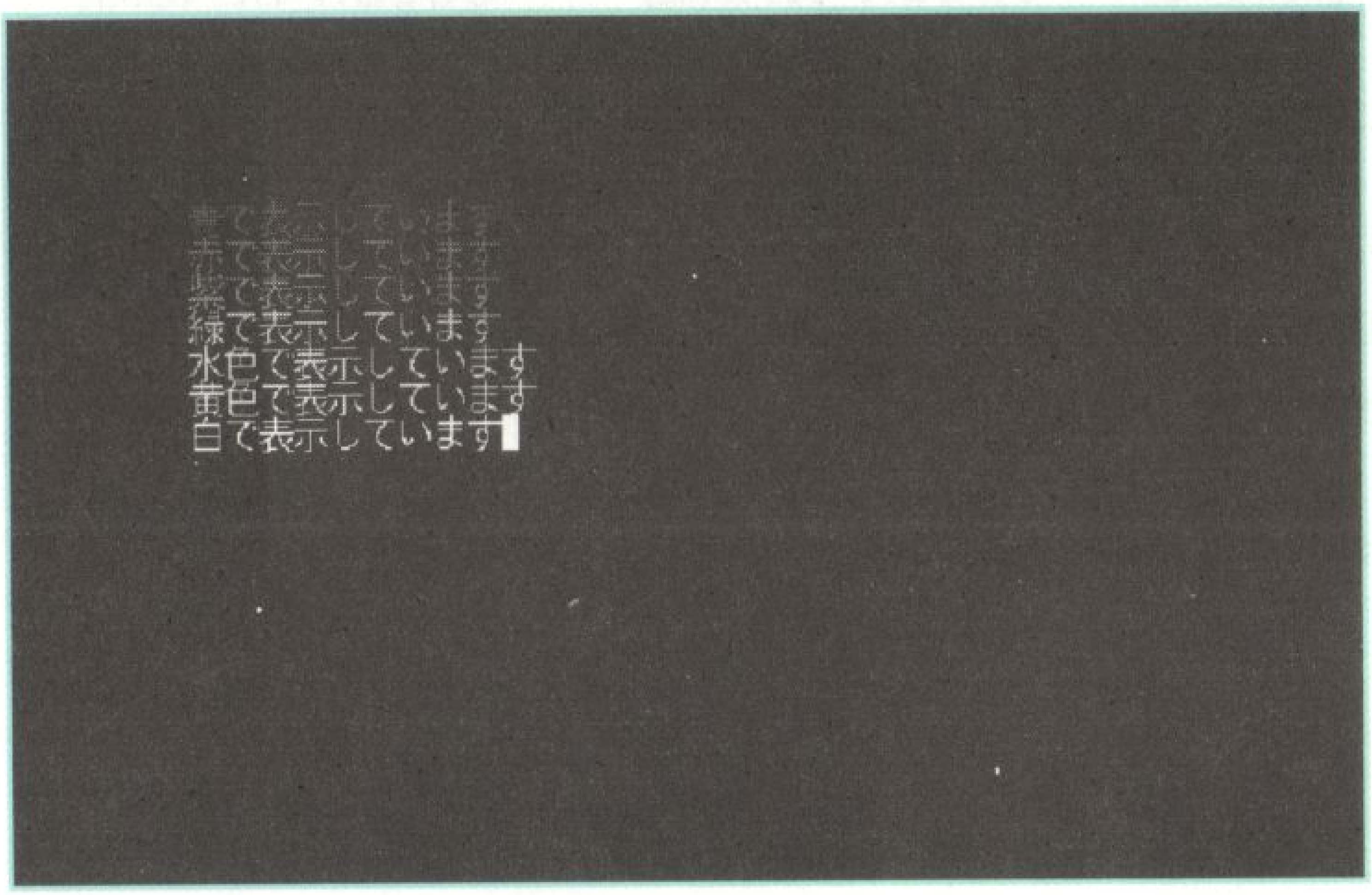
    move(11, 10);
    attrset(COLOR_PAIR(WHITE));
    addstr("白で表示しています");

    refresh();
    getchar();
    endwin();

    return 0;
}
```

【実行例】

図32-3▶



次にプログラムのポイントを説明する．まず，以下はカラーペアを番号ではなく名前で参照できるように記号定数を定義しているところである．

```
#define BLACK       0
#define BLUE        1
#define RED         2
#define PURPLE      3
#define GREEN       4
#define CYAN        5
#define YELLOW      6
#define WHITE       7
```

色を使えるようにするため，initscrの後でstart_colorを実行している．

```
initscr();
start_color();
```

次が使用するカラーペアを定義しているところである．カラーペアの0は定義できないので注意．

```
init_pair(BLUE,     COLOR_BLUE,     COLOR_BLACK);
init_pair(RED,      COLOR_RED,      COLOR_BLACK);
init_pair(PURPLE,   COLOR_MAGENTA,  COLOR_BLACK);
init_pair(GREEN,    COLOR_GREEN,    COLOR_BLACK);
init_pair(CYAN,     COLOR_CYAN,     COLOR_BLACK);
```

```
init_pair(YELLOW,  COLOR_YELLOW,   COLOR_BLACK);
init_pair(WHITE,   COLOR_WHITE,    COLOR_BLACK);
```

それぞれの文字は，次のようにして表示している．

```
move(5, 10); ―――――――――――― 表示位置にカーソルを移動
attrset(COLOR_PAIR(BLUE)); ―――― 色を指定
addstr("青で表示しています");――― 表示
```

もちろん，refresh関数を実行しないと表示されないので，最後にまとめてリフレッシュする．

```
refresh();
```

第33章 スクリーンエディット

この章では，**スクリーンエディタ**を開発する．

スクリーンエディタというと，かなり大きなプログラムになる．リストだけでも膨大な量になる．したがって，この章だけでその全体を掲載することはできない．1章分程度に収まるように縮小してある．しかし，小さなプログラムではあるが，スクリーンエディタを開発する上で基本となる考え方は織り込んである．

- **C++で開発するのだからクラスにまとめた**
- **イベント駆動型のアプリケーションに組み込める形にまとめた**
- **日本語に対応している**

この章で示す小さなエディタを発展させて，大きなスクリーンエディタを完成させることも可能である．

1 どのようなエディタか？

最初に，次の画面を見ることから始める．

図33-1▶



これは，あるアプリケーションのデータ入力用の画面である．右側に入力用の領域があり，左側にその入力項目の内容を示すラベルが表示されている．この各項目でデータを入力するとき，スクリーンエディットができると修正が楽である．

この章で作成するのは，このような項目の入力に適した**編集ボックス**である．編集ボックスをただ表示したのでは，何を入力するのかわからない．そこで，その項目の内容を表示する**ラベル**もいっしょに表示することが多い．そこで，これから作成する編集ボックスも，ラベル付きのものにする．

2 イベント駆動型プログラミング

この章で作成する編集ボックスは，**イベント駆動**型のアプリケーションに組み込める形にまとめてある．

「イベント駆動型のプログラム」というのは，プログラムの実行制御がイベントの発生を中心に動く形式のプログラムをいう．ここで**イベント**というのは，プログラムが認識する「出来事」のことである．

- **キーが押された**
- **マウスがクリックされた**
- **メニューコマンドが実行された**
- **指定の時間になった**
- **その他いろいろ**

これらのイベントが発生すると，決められた動作が実行される．たとえば，ある項目の上でキーが押されると，「キーが押された」というイベントがその項目に渡される．その項目は，入力されたキーを自分なりに処理する．

どのようなイベントがどのような順序で発生するかは，そのプログラムを実行しているユーザー次第である．イベント駆動型のプログラムは，プログラムが実行される順序をプログラム時に決める必要がないので，その制御をユーザーの気ままに委ねることができる．

また，イベント駆動型のプログラムでは，「イベントを送信するプログラム」と「イベントを受信するプログラム」を分けて書くことができる．このようにイベント処理を分けることにより，それぞれの部品は，自分に関係のあるイベントの処理に専念することができる．

このようにイベント駆動型のプログラムは，対話型のアプリケーションを作成するのに向いている．また，オブジェクト指向プログラミングとも，たいへんに相性がよい．

3 クラスの設計

これから編集ボックスの設計に入る．1行単位のスクリーンエディタといっても，それなりの大きさになることが予想される．せっかく作るのだから，後で再利用できる形にまとめたい．すると，次の2つの選択肢が存在する．

編集ボックスを「関数」の形にまとめる
編集ボックスを「クラス」の形にまとめる

もし，このプログラムをC言語で作成するなら，選択の余地はない．前者で決まりである．C言語はクラスを使うことはできない．しかし，C++はどちらも利用できるので，選択に迷うところである．

3.1◆クラスにまとめる

編集ボックス程度のプログラムになると，サイズの上からも全体を1つの関数にまとめるのは困難である．自然と複数の関数で構成することになる．

それらの関数は，「入力バッファ」とか，「カーソル位置」とかの情報を表現する変数を共有する必要がある．「グローバル変数を複数の関数で共有する」という関係にあるとき，それらの変数や関数を1つのクラスにまとめると管理しやすい．

また，編集ボックスで使われる関数は，利用する順序に規則性が存在する可能性がある．たとえば，大雑把に考えても，次のような関数が必要になる．そして，これらの関数は，次の順序で呼び出す必要がある．順序を変えると，うまく動かない．

編集ボックスを初期化する関数を最初に実行する
↓
入力を行う編集ボックスにフォーカスを当てる関数を実行する
↓

編集を行う関数を実行する

↓

後処理を行う関数を実行する

このように関数の使い方に規則が存在する場合，それらの関数を1つのクラスにまとめることで，使い方を隠蔽することができる．

いずれにしても，せっかくC++で開発するのだから，練習のためにもクラスにまとめるのがよさそうである．ということで，次のようなクラスを作成する．

```
class EditBox {
        …………
};
```

編集ボックスクラスの名前をEditBoxにした．あとは，このクラスの中身を作れば完成である．といっても，それが大変なのであるが…．

3.2◆座標

EditBoxクラスに必要なメンバー変数は何だろう？　データの持たせ方によって，プログラムのしやすさが変わってくる．メンバー変数の設計は，十分に検討する必要がある．

で，必要な変数を，次の2種類に分けて考えていく．

- **表示上必要なデータを保持するためのメンバー変数**
- **編集作業に必要なデータを保持するためのメンバー変数**

最初に「表示上必要なデータ」から考えていく．

この編集ボックスは，表示上，次の2つのものを持っている（表示しなければならない）．

ラベル
編集領域

まず必要なのが，どこに表示するかを示す**座標**である．

座標については，ラベルの位置が決まれば，編集領域の座標は計算で求めることができる．しかし，ラベルの方は，一度表示してしまえば後はいじくる必要がない．それに対して，編集領域の方は，時々刻々と描き直さなければならない．

したがって，編集領域にも座標を持たせるほうが，プログラムしやすくなりそうである．次のメンバー変数を用意することにする．

編集領域の座標= (home_x, home_y)

ラベルの方の座標は，x座標だけを保持する．y座標の方は，編集領域の座標と同じなので，共有することができる．

ラベルの座標= (label_x, home_y)

というわけで，ここまで決まったメンバー変数をEditBoxに挿入すると，次のようになる．

```
class EditBox {
    int label_x;                // ラベルの座標
                                // y座標は編集領域と同じ

    int home_x;                 // 編集領域の座標
    int home_y;
    ............
};
```

3.3◆表示するもの

座標だけでは，表示できない．表示すべき「中身」が必要である．ラベルを表示するのに必要なデータは，文字列である．それを次のメンバー変数で表現する．

```
char label[20];
```

この場合，ラベルの最大文字数は19文字（最後に“\0”が必要なので20文字ではない）までになる．必要なら，もう少し大きく取ってもよい．具体的な文字列は，コンストラクタで取得すればよいだろう．

編集領域の方は，ユーザーがデータを入力するまでは表示すべき内容がない．しかし，編集領域を表示したほうが入力しやすいので，入力可能文字数を，次のメンバー変数で表現することにする．

```
int width;
```

新しく決まったメンバー変数をEditBoxに追加すると次のようになる．

```
class EditBox {
      int label_x;                   // ラベルの座標
                                     // y座標は編集領域と同じ
      char label[20];                // 編集領域の左に表示するラベル

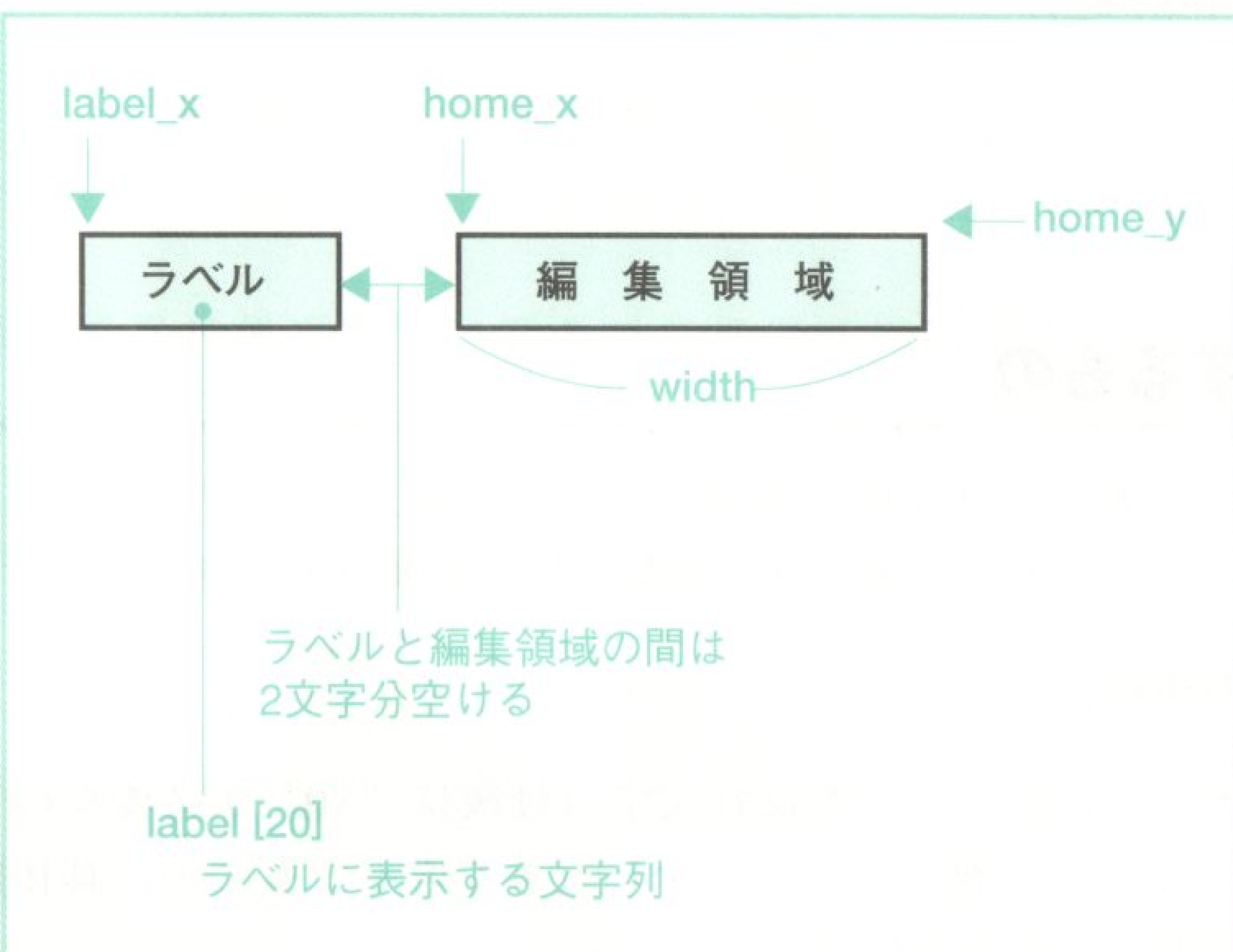
      int home_x;                    // 編集領域の座標
      int home_y;

      …………

};
```

また，ここまでで表示上必要なデータは出揃ったので，次に，それらの変数を図式化しておく．このような図を用意しておくと，プログラムするとき，いちいちリストのあちこちを見直さなくても済むので楽である．

図33-2▶



3.4◆編集バッファ

スクリーンエディタには，表示するもの以上に必要な重要なデータがある．**編集バッファ**である．

「編集バッファ」というのは，入力されたデータを保存する場所である．編集

ボックスのように1行だけ編集できればよいというのであれば，バッファの構造は単純である．しかし，何行にもわたるスクリーンエディタでは，バッファの構造がプログラムの性能や作りやすさにも影響に与える．バッファの構造はスクリーンエディタの命なのである．

このプログラムは，単純な1行エディタなので，編集バッファは単純な文字の配列にする．ただし，メンバー変数としてはchar *のポインタを用意するだけにする．

```
char *buf;
```

bufが編集バッファの先頭アドレスを記憶する．ただし，このままでは編集バッファのためのメモリが確保されていない．コンストラクタなどでメモリを確保する必要がある．

なぜこのような構造にしたかというと，編集領域のサイズを自由に設定できるようにしたかったからである．コンストラクタの引数に編集バッファを指定してもらうようにすれば，個々の編集ボックスごとに編集領域のサイズを変更することができる．

編集バッファでは，もう1つ，次の変数が必要である．

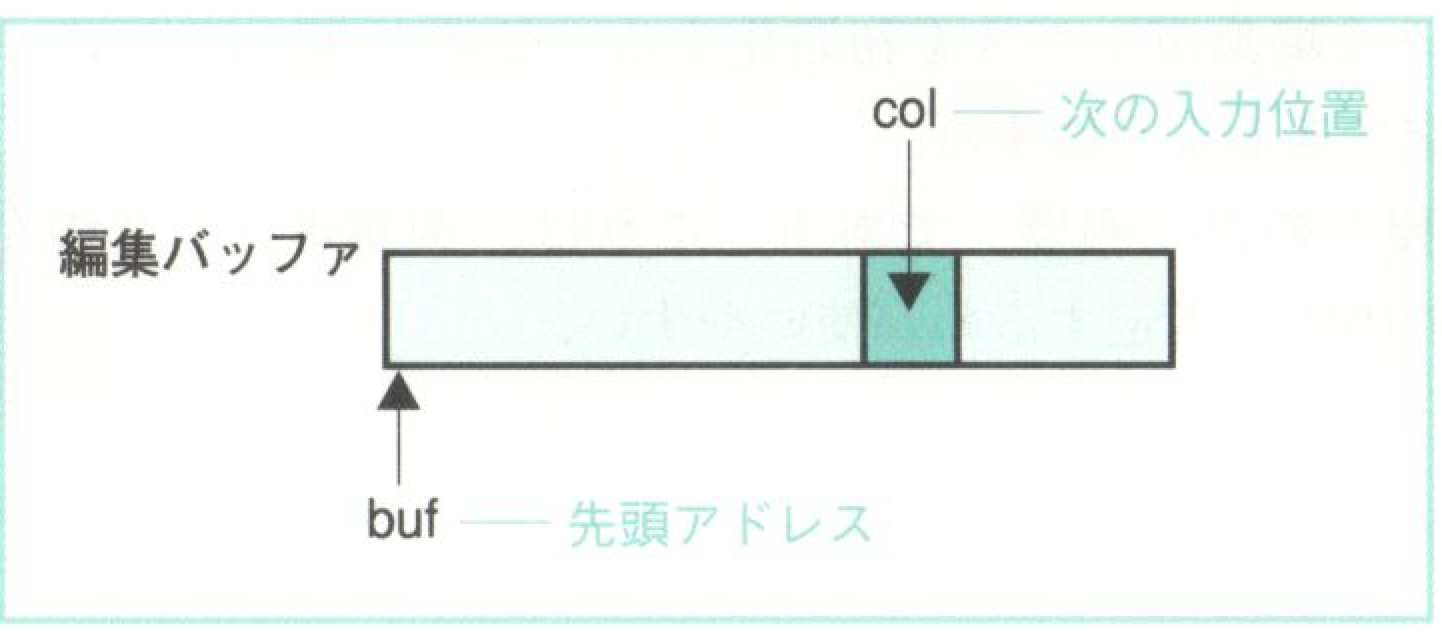
```
int col;
```

colは，buf上の次の入力位置を指すものである．初期値は0である．1文字入力するごとにカウントアップしていく．逆に，1文字削除されれば，カウントダウンする．また，カーソル移動でもcolの位置は移動する．

このようにcolが次の入力位置を覚えておけば，キー入力された文字を，簡単にバッファに挿入することができる．

これで，編集バッファのデータが揃った．次にそれを図示しておく．

図33-3▶



3.5◆メンバー変数

ここまでの考察で，すべてのメンバー変数が出揃った．次に，ここまでのクラス宣言をまとめておく．

```
class EditBox {
        int label_x;                    // ラベルの座標
                                        // y座標は編集領域と同じ
        char label[20];                 // 編集領域の左に表示するラベル

        int home_x;                     // 編集領域の座標
        int home_y;

        char *buf;                      // 編集バッファ
        int col;                        // buf上の次の入力位置
        ............
};
```

4 コンストラクタとデストラクタ

次に，EditBoxで外部公開するメンバー関数（publicセクションに置くメンバー関数）を検討していく．これにより，EditBoxの使い勝手が決まる．最初は，最も重要なメンバー関数であるコンストラクタとデストラクタについてである．

4.1◆コンストラクタの設計

コンストラクタで編集ボックスを初期化する．そのとき，編集ボックスの属性を「引数」で指定できるようにする．

引数でまず必要なのが「座標」である．これは，編集ボックスの左端の座標，すなわちラベルの座標で指定するのが使いやすい．

```
EditBox(int x, int y,…);
```

ラベルの座標

さらに，ラベルに表示する文字列も必要である．これは，文字列で指定する．

```
EditBox(int x, int y, char *l,…);
```

ラベルに表示する文字列

最後に編集領域のサイズも必要である．

```
EditBox(int x, int y, char *l, int w);
```

編集領域のバイト数

これでコンストラクタの仕様が決まったので，EditBoxのクラス宣言にコンストラクタを追加する．

```
class EditBox {
    int label_x;
    …………
    int col; ──────────ここまでメンバー変数
public:
    EditBox(int x, int y, char *l, int w); ────コンストラクタ
};
```

4.2◆コンストラクタの定義

ここまで設計ができるとコンストラクタ本体の定義が可能である．最初にラベル部分の初期化を考える．これは，座標と表示する文字列をメンバー変数に保存するだけでよい．

```
label_x = x; ──────ラベルのx座標
strcpy(label, l); ──────ラベルに表示する文字列（エラー処理は省略）
```

編集領域については，メンバー変数の初期化と編集バッファ用のメモリの確保が必要である．編集領域のx座標であるが，これは，ラベルの位置から次のように

して計算する必要がある．

ラベルのx座標＋ラベルの文字数＋2

最後に2を足しているのは，ラベルと編集領域の間を2文字分空けるためである．これをプログラムで書くと次のようになる．

```
home_x = x + strlen(l) + 2;
```

そのほかにメンバー変数の初期化で必要なものを次に示す．

```
home_y = y;  ――― 編集領域のy座標
width = w;   ――― 編集領域のサイズ
col = 0;     ――― 次の入力位置
                  最初はバッファの先頭
```

最後に編集領域のためのメモリの確保をする．サイズはwidth + 1である．1を足すのは，文字列の最後に“\0”を置く必要があるからである．

```
buf = new char[width + 1]; ――― メモリを確保
*buf = '\0';               ――― 最初は空の文字列にする
```

以上でコンストラクタのコードが出来上がった．次に，そのリストを示す．

```
EditBox::EditBox(int x, int y, char *l, int w)
{
    // ラベルの初期化
    label_x = x;
    strcpy(label, l);

    // 編集領域の初期化
    home_x = x + strlen(l) + 2;
    home_y = y;
    width = w;
    col = 0;

    buf = new char[width + 1];
    *buf = '\0';
}
```

4.3◆デストラクタ

コンストラクタの中で編集用のメモリを確保したので，その領域をデストラクタで解放する．

```
class EditBox {
      int label_x;
      …………
      int col;
public:
      EditBox(int x, int y, char *l, int w);
      ~EditBox(); ――――――――――――デストラクタ
};
```

デストラクタ本体の定義は次のようになる．

```
EditBox::~EditBox()
{
      delete [] buf; ――――――――――――編集バッファを解放する
}
```

4.4◆なぜcursesの設定をしないのか？

いま設計してきたコンストラクタとデストラクタを見て，cursesの初期化と後処理を組み込んでいないのを不思議に思ったかもしれない．たとえば編集ボックスを使うためには，cursesの初期化が必要である．

```
initscr(); ――――――――――――cursesの使用開始
noecho(); ――――――――――――キー入力時に表示しない
raw(); ――――――――――――rawモードにする
```

これらのコードをコンストラクタの中に入れておけば，毎回mainに書く必要がなくなる．また，書き忘れもない．同様に，デストラクタの中に次のコードを入れておけば，cursesの後処理も簡単である．

```
endwin();
```

しかし，EditBoxのコンストラクタとデストラクタは，そのようにしなかった．

なぜか？――それは，EditBox以外の部品を使うかもしれないし，複数の編集ボックスを使うかもしれないからである．

もし，1つのアプリケーションで1つの編集ボックスしか使わないのなら，そのような設計でもかまわない．しかし，入力項目がいくつかあれば，複数の編集ボックスが必要になる．それらの編集ボックスを使うたびにコンストラクタが呼び出される．すると，cursesの初期化が何回も行われることになってしまう．

やはりcursesの初期化と後処理は，EditBoxの外で行ったほうがよいのである．

5 公開メンバー

コンストラクタ／デストラクタ以外で外部に公開するメンバー関数を検討する．

5.1◆表示

最初は簡単なところからshowを取り上げる．第4.2節のコンストラクタの定義を見ればわかるように，コンストラクタを起動しただけでは編集ボックスは表示されない．そこで，showというメンバー関数を用意し，この関数が実行されると編集ボックスを表示するようにする．

```
class EditBox {
        …………
public:
        EditBox(int x, int y, char *l, int w);
        ~EditBox();
        void show(); ―――――――――― 追加する
};
```

ここまでのEditBoxの使い方は次のようになる．

```
EditBox eb(10, 5, "文字列編集:", 10);
        ―――――――――― 編集ボックスebを作成
                    ここでコンストラクタが実行される
eb.show(); ―――――――――― ebを表示する
```

showの定義は簡単である．ラベルを表示し，その右側に編集領域の文字数分のアンダーラインを表示する．

```
// エディットボックスを表示する
void EditBox::show()
{
    attrset(WA_NORMAL);
    move(home_y, label_x);
    addstr(label);

    attrset(WA_UNDERLINE);
    cls(); ―――――――――――――― これは何？
}
```

showの定義の最後にあるclsは，編集領域の表示をクリアするものである．この関数を外部に公開する必要はないので，メンバー変数と同じように，privateセクションで宣言している．

```
class EditBox {
    ............
    void cls(); ―――――――――――― 非公開メンバー関数
public:
    EditBox(int x, int y, char *l, int w);
    ~EditBox();
    void show();
};
```

clsの定義は次のようになる．

```
// 編集領域をスペースでクリアする
// バッファには手をつけない
void EditBox::cls()
{
    attrset(WA_UNDERLINE);
    move(home_y, home_x);
    for (int i = 0; i < width; i++)
        addch(' ');
```

```
        refresh();
    }
```

5.2◆フォーカスを当てる

1つのアプリケーションで使われる編集ボックスの数が1つだけとは限らない．複数の項目を編集するのであれば，その数だけの編集ボックスが使われる．すると，**フォーカスの移動**ということを考慮する必要が出てくる．

フォーカスとは，カメラの「焦点」のことである．フォーカスの当てられた編集ボックスが，現在入力可能な編集ボックスである．フォーカスは，必要に応じて次々と編集ボックスの上を移動していく．そこで，次の公開メンバーを用意する．

```
class EditBox {
    ............
public:
    ............
    void setfocus(); ────────── 追加する
};
```

setfocusには，その編集ボックスにフォーカスが移動したときに必要な処理を書く．このプログラムでは，カーソルを先頭に戻し，アトリビュートをアンダーラインに設定している．

```
// フォーカスを得たときの処理
void EditBox::setfocus()
{
    move(home_y, home_x);
    attrset(WA_UNDERLINE);
    refresh();
}
```

プログラムを拡張する場合は，この関数を書き換えたり，処理を追加したりすることになる．

5.3◆イベントハンドラ

編集ボックスをイベント駆動型のアプリケーションフレームワークに組み込む

とき，編集ボックスはイベントの「受け取り側」になる.「送り側」ではない. イベントの検出と配送は，アプリケーションフレーム側の仕事になる.

そこで，編集ボックスにイベントを受け取るための関数を用意する. イベントを受け取り，それを処理するための関数を**イベントハンドラ**という.

```
class EditBox {
    ............
public:
    ............
    void handle_event(int e); ———— イベントハンドラ
};
```

eが受け取るイベントである. このプログラムでは，イベントといってもキー入力しか扱わない. したがって，eの型としてはintで十分である. しかし，もう少し大きなイベント駆動型のプログラムでは，イベントの型ももう少し複雑になる.

編集ボックスでは，handle_eventが中心的な関数になる. なぜなら，この関数がスクリーンエディタとしての入口であり，スクリーンエディタとしてのメインプログラムになるからである. この関数を完成させれば，スクリーンエディタも完成する.

5.4◆編集結果

最後に，編集ボックスで編集された文字列を受け取るための公開関数を用意する.

```
class EditBox {
    ............
public:
    ............
    char *get_text();
};
```

たとえば，次のようにすれば編集したテキストを表示することができる.

```
EditBox eb(10, 5, "文字列編集:", 10);
............
addstr(eb.get_text());
```

get_textの定義は次のようになる．

```
// 現在のテキストを返す
char *EditBox::get_text()
{
    return buf;
}
```

6 クラスの使い方

これで，コンストラクタ／デストラクタを含むすべての公開メンバーが出揃った．ということは，（もし，その中身のコードが完成していれば）EditBoxクラスを使うことができる．次にEditBoxクラスの使用例を示す．

```
int main()
{
    int e;

    initscr();  ──── cursesの初期化
    noecho();
    raw();

    EditBox eb(10, 5, "文字列編集:", 10);  ──── 編集ボックスの作成
    eb.show();
    eb.setfocus();

    while ((e = getevent()) != CTRL_Q)  ──── イベントループ
        eb.handle_event(e);

    move(7, 10);  ──── 結果の表示
    attrset(WA_NORMAL);
    addstr("編集結果    :  ");
    attrset(WA_UNDERLINE);
    addstr(eb.get_text());
    move(11, 10);
    attrset(WA_REVERSE);
    addstr("何かキーを押すと終了");
```

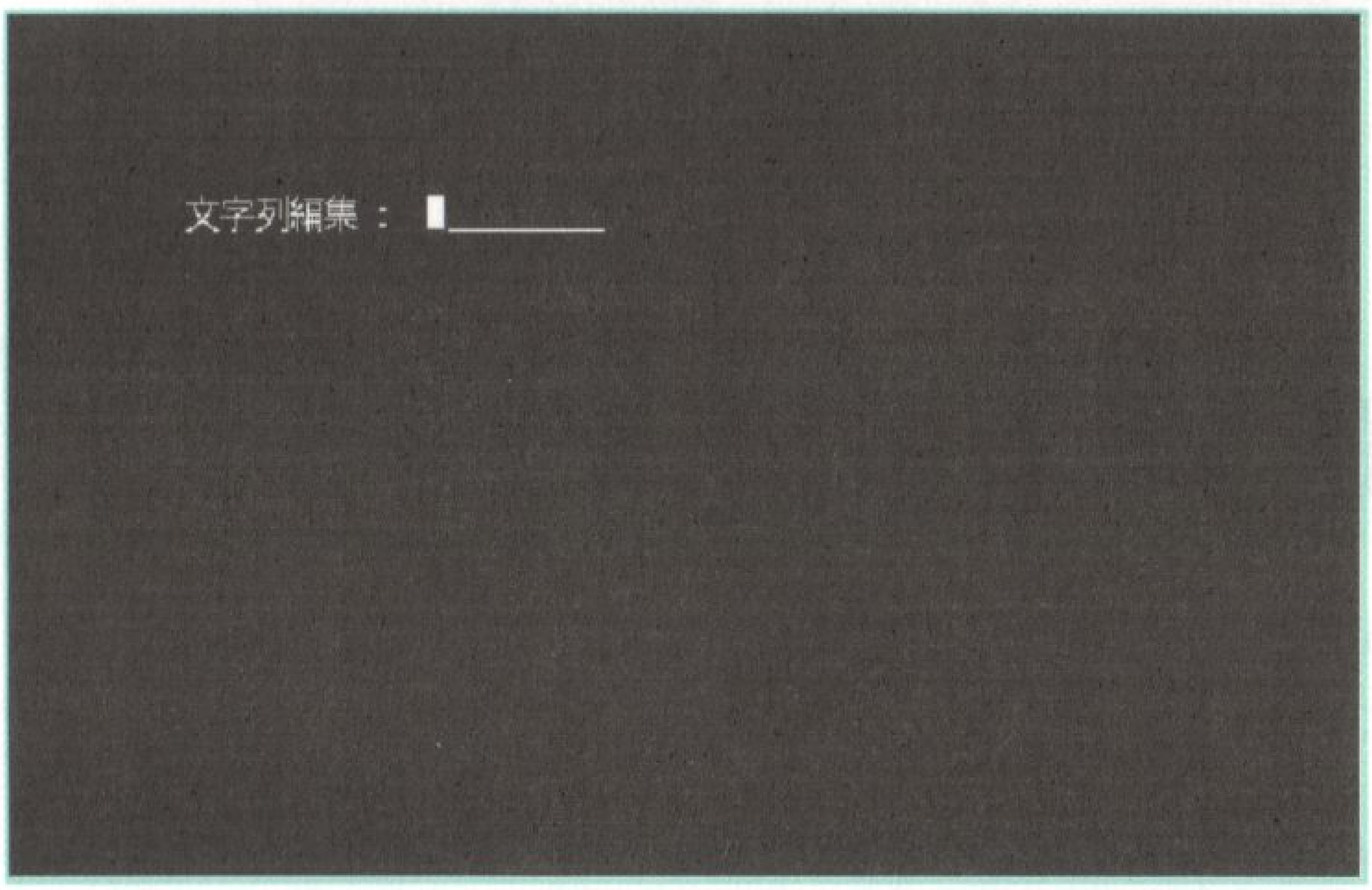
```
    refresh();
    getchar();

    endwin(); ―――――――――――――――― cursesの後処理

    return 0;
}
```

このプログラムは，関数mainを示しただけであるが，もしこれにEditBoxクラスの定義を追加すれば，次のように実行することができる．

図33-4▶

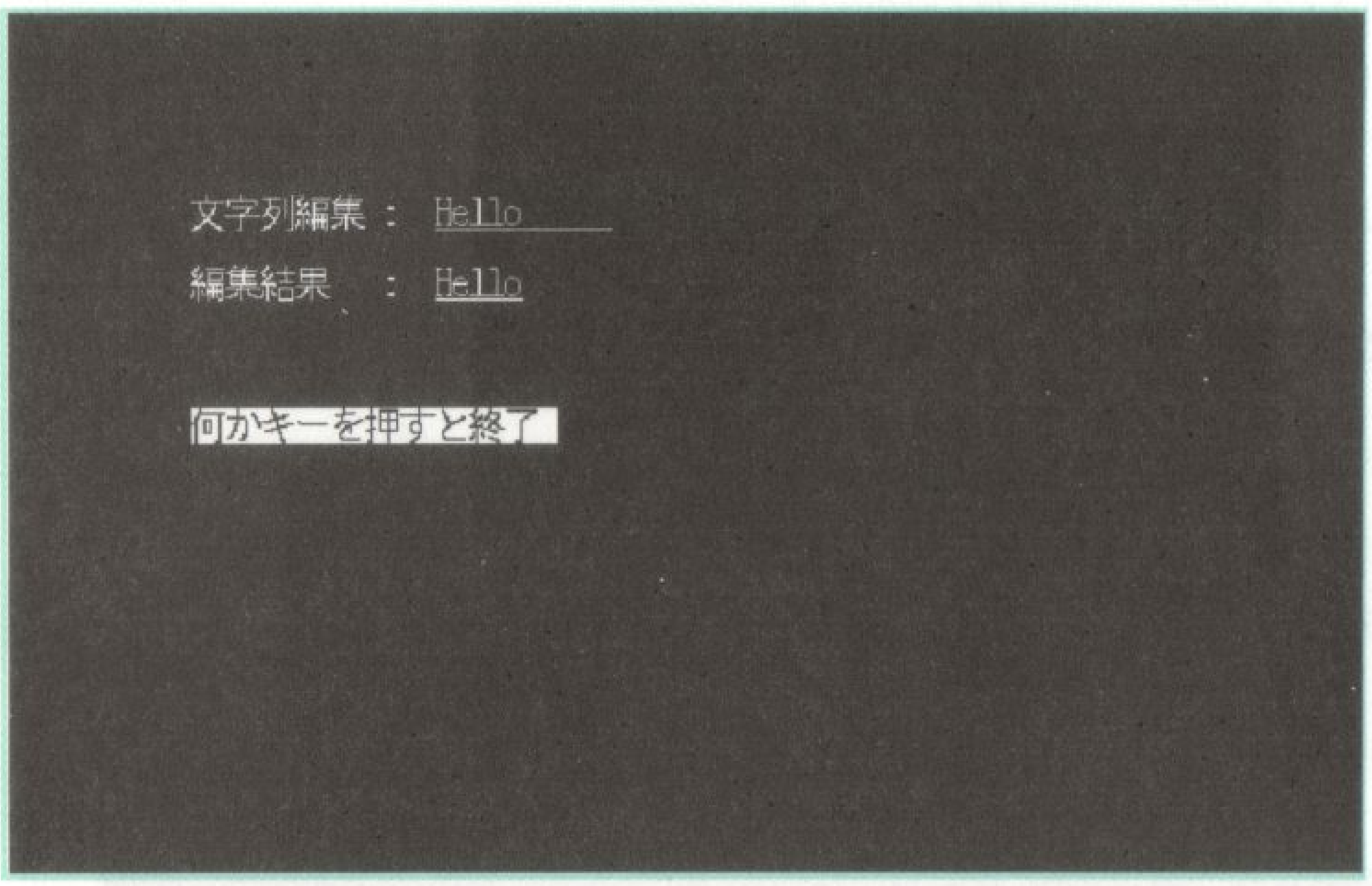


実行すると，「**文字列編集:**」というラベルの付けられた編集ボックスが表示される．編集ボックスでは，次のキーを使って編集することができる．

キー	機能
CTRL_B	カーソルを左に移動する
CTRL_F	カーソルを右に移動する
CTRL_D	カーソルの上の1文字を削除する
CTRL_H	カーソルの右の文字を削除する
CTRL_U	編集領域をクリアする （すべての文字を消す）
CTRL_Q	編集を終了する

適当な文字列を編集して Ctrl ＋ Q を押せば，編集ボックスの下に，いま入力した文字列が表示される．

図33-5▶



さらに適当なキーを押せば，プログラムが終了する．

7 イベントの検出

第6節のmainプログラムでは，次のループでイベントを検出し，それを編集ボックスに配送していた．

```
while ((e = getevent()) != CTRL_Q)
      eb.handle_event(e);
```

ここで，geteventがイベントを検出する関数である．この関数は，（メンバー関数ではなく）一般関数として作成してある．以下に，そのリストを示す．

```
int getevent()
{
      int c, c2;

      if ((c = getchar()) > 0x7f) {
            c2 = getchar();
            return (c << 8) | c2;
      }
      else
            return c;
}
```

geteventは，イベントを検出し，それを返す関数である．このプログラムでは，イベントはキー入力だけである．したがって，geteventの関数値は，入力したキーのコードである．しかし，その割には凝ったことをしている．これは，geteventが漢字の処理を行っているからである．

geteventが返す値（イベント）は，次のようになっている．

文字の種類	geteventの値
非漢字	キーコード
漢字	上位バイト=漢字の1文字目
	下位バイト=漢字の2文字目

つまり，イベントの上位バイトを見て，それが0なら非漢字，0でなければ漢字と判断することができる．

geteventの内部をもう少し詳しく説明する．

geteventは，漢字コードとしてUNIXの**EUC**を想定している．EUCは，漢字の扱いが簡単である．最上位のビットが1なら漢字コードの一部と判定できるからである（半角のカナは処理していない）．

ということで，geteventの処理は次のようになっている．

① getcharで1文字入力する
② もしそれが漢字ならば（0x7fよりも大），もう1文字入力して2バイトにして返す
③ そうでなければ非漢字なので，そのまま返す

8 スクリーンエディタの内部構造

付随部分の説明が終わったので，いよいよ核となるスクリーンエディタの中身の解剖に入っていく．これは，公開メンバーのhandle_eventを完成させることにほかならない．

8.1◆処理する機能

この編集ボックスが処理する機能は次の6つである．

キー	機能
CTRL_B	カーソルを左に移動する
CTRL_F	カーソルを右に移動する
CTRL_D	カーソルの上の1文字を削除する
CTRL_H	カーソルの右の文字を削除する
CTRL_U	編集領域をクリアする (すべての文字を消す)
CTRL_Q	編集を終了する

そこで，これらの6つのコントロールキーを検出できるように，あらかじめ定数を定義しておく．

```
const int CTRL_B = 0x02;
const int CTRL_D = 0x04;
const int CTRL_F = 0x06;
const int CTRL_H = 0x08;
const int CTRL_Q = 0x11;
const int CTRL_U = 0x15;
```

これで，handle_eventが受け取ったイベントがCtrl_Bなら Ctrl ＋ B が押されたものと判定することができる． Ctrl ＋ B を押すと，どのコードが生成されるかは，システムによって異なる可能性がある．第32章第7.2節で紹介したようなプログラムを作って調べてみるとよい．

handle_eventのコードは，次のような感じになる．

```
void EditBox::handle_event(int e)
{
    if (e == CTRL_B)
        // カーソルを左に移動する
    else if (e == CTRL_F)
        // カーソルを右に移動する
        …………
    else
        // 1文字挿入追加
}
```

すなわち，入力された文字を調べて，それが有効なコントロールキーならそれを処理する．また，コントロールキー以外なら普通の文字なので，編集ボックスに挿入する．

このようなことを書けばよいわけである．ただし，それぞれの処理は大きくなる可能性があるので，下請けのメンバー関数を作って，それに処理を任せることにする．

次にhandle_eventの完成版を示す．

```
void EditBox::handle_event(int e)
{
    if (e == CTRL_B)
        moveleft();  ―――― カーソルを左に移動
    else if (e == CTRL_F)
        moveright(); ―――― カーソルを右に移動
    else if (e == CTRL_H)
        bschar();    ―――― 左の文字を削除
    else if (e == CTRL_D)
        delchar();   ―――― カーソル上の文字を削除
    else if (e == CTRL_U)
        empty();     ―――― 編集領域を空にする
    else if (e < 0x20)
        beep();      ―――― その他のコントロールコードは無効にする
    else
        inschar(e);  ―――― 1文字挿入追加
    refresh();
}
```

ここで，moveleft，moverightなどが下請けのメンバー関数である．これらの関数は公開する必要がないので，privateセクションで宣言する．

```
class EditBox {
    ............
    void moveleft(); ―――― privateメンバー関数
    void moveright();
    void bschar();
```

```
        void delchar();
        void empty();
        void inschar(int e);
public:
        ............
        char *get_text();
};
```

8.2◆カーソル移動

これからhandle_eventの下請け関数を作成していく．最初はカーソルを移動するためのmoveleftとmoverightである．

moveleft ―――― カーソルを1文字分左に移動する
moveright ―――― カーソルを1文字分右に移動する

カーソル移動に必要な処理は，次の2つである．

- 編集バッファ上の入力位置を示すcolの更新
- 実際にカーソルを移動する

このうち，実際にカーソルを移動する部分は，colの位置が決まれば，cursesのmove関数を使って，次のように書くことができる．

```
move(home_y, home_x + col);
```

colの方は，漢字を扱うので少しややこしい．moveleftを例に考える．この関数はカーソルを1文字左に移動する．ゆえに，単純には次のようにすればよい．

```
--col;
```

ただし，colがバッファの左端にいる場合（これはバッファが空の状態を指す）は，カーソルが移動できない．そこで，次のようなエラーチェックを入れる．

```
if (col <= 0)
        beep(); ―――― 左端の場合はベルを鳴らす
else
        // 1文字左に移動
```

このチェックを通過した後は，カーソルの左には少なくとも1文字は存在するこ

とになる．ただし，それが漢字かどうかで処理が変わる．漢字の場合は2文字分左に移動しなければならないからである．

というわけで，moveleftのコードは次のようになる．

```
void EditBox::moveleft()
{
    if (col <= 0) ―――― バッファの左端にいるときは移動できない
        beep();
    else if (buf[--col] & 0x80) {
        --col; ―――― 漢字の場合
        move(home_y, home_x + col);
    }
    else ―――― 残りは非漢字
        move(home_y, home_x + col);
}
```

同様に，moverightも次のように出来上がる．

```
void EditBox::moveright()
{
    if (!buf[col]) ―――― 最後の文字の右にいるときは移動できない
        beep();
    else if (buf[col++] & 0x80) {
        col++;; ―――― 漢字の場合
        move(home_y, home_x + col);
    }
    else ―――― 残りは非漢字
        move(home_y, home_x + col);
}
```

8.3◆編集バッファの構造

次に，文字を挿入する関数**inschar**を考える．これは（漢字をサポートしようとすると）かなりややこしい．

一般にエディタで文字を挿入するには，次の2つの処理が必要である．

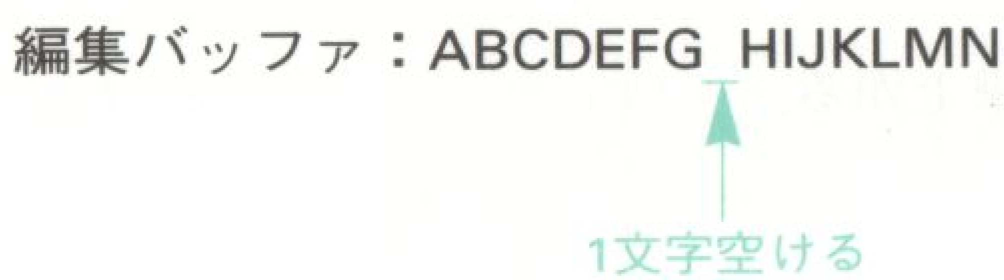
- 入力された文字を編集バッファの適切な位置に挿入する
- それを画面に反射する

まず，前者から考える．**編集バッファ**というのは，エディタに入力されているすべての文字を保存する場所である．編集バッファの構造は，単純なものから複雑なものまで，さまざまである．

最も単純な編集バッファは，入力された文字をそのまま並べただけという構造のものである（このプログラムでは，この単純な構造を採用する）．たとえば，いま“ABCDEFGHIJKLMN”という文字列がエディタに入力されているとする．すると，編集バッファの状態は次のようになる．

編集バッファ：ABCDEFGHIJKLMN

この状態で文字“4”が入力されたとする．また，現在のカーソルの位置が“H”の上にあるとする．すると，新しい文字を挿入するためには，Hから後ろの部分全体を1文字分後方に移動する必要がある．



そして，空けた位置に入力された文字“4”を挿入する．



これが単純な構造を持つ編集バッファにおける文字の挿入である．この構造の長所は，構造が単純であるがゆえにプログラムが楽であるということである．しかし，速度の面では芳しくない．この例では，挿入位置から後ろにわずか7文字しかない．しかし，巨大なファイルの先頭部分に1文字挿入しようとすると，移動に時間がかかってしまう．この欠点を補うためには，より複雑な構造の編集バッファを用意しなければならない．

この章のエディタは，1行だけの簡単な編集ボックスである．そこで，この単純な構造をそのまま採用することにする．

8.4◆画面への反射

入力された文字を編集バッファに挿入しただけでは，それが画面に反映されない．編集バッファの内容を画面に反映させる必要がある．編集バッファの状態を画面に反射させて，初めて編集結果が目に見える形になる．

複数の行を持つ通常のエディタでは，これがけっこう大変な処理になる．文字列の並びという「1次元構造」のバッファをスクリーンという「2次元構造」の画面にマップしなければならない．もし行末に漢字が来たら，1バイト目と2バイト目が泣き別れないように面倒も見なければならない．考えるだけでも大変そうである．

この処理をどう行うかは，編集バッファの構造で決まる．設計の面白いところでもある．

一方，編集ボックスのような1行エディタなら，この処理も簡単である．編集バッファを文字列と見なして，それを表示するだけである．

```
move(home_y, home_x);            ――― 編集バッファの先頭に移動して
addstr(buf);                     ――― 編集バッファの内容を表示
move(home_y, home_x + col);      ――― カーソル位置を現在位置に戻す
```

8.5◆ASCII文字の挿入

第3.3節～第3.4節で「文字を挿入する」考え方を説明してきた．それを具体的なプログラムにしていく．つまり，次の関数を完成させていく．

```
void EditBox::inschar(int e)
{
    // イベントeに入っている文字を挿入する
}
```

この処理は，最初に警告したとおり，かなりややこしい．しかし，1つ1つの処理は単純なので，一歩一歩つぶしていけばよい．

まずプログラムの前半（というか大半）は，文字をバッファに挿入することである．そして，後半がバッファの内容を画面に表示することである．

```
void EditBox::inschar(int e)
{
    // 文字を編集バッファに挿入する
    // 編集バッファの内容を編集領域に表示する
}
```

後半の部分は，すでに第3.4節の最後のところで示した．

```
void EditBox::inschar(int e)
{
    // 文字を編集バッファに挿入する

    // 編集バッファの内容を編集領域に表示する
    move(home_y, home_x);
    addstr(buf);
    move(home_y, home_x + col);
}
```

前半部分は，まず入力された文字が漢字かどうかで処理が2つに分かれる．

```
if (e > 0x7f)
    // 漢字の場合の処理
else
    // ASCII文字の場合の処理
```

ASCII文字――すなわち非漢字のほうが処理が簡単なので，先にASCII文字の場合を考える．まずチェックしなければならないのが，文字を入力する場所が残っているかどうかである．すでに編集領域いっぱいに文字が入力されていたら，もうそれ以上文字は入力できない．その場合は，ベルを鳴らして警告する．

```
if (strlen(buf) >= width) {  ――― 編集バッファがいっぱいなら
    beep();  ――― ベルを鳴らす
    return;
}
else {
    ←――― そうでなければ文字を挿入する
}
```

文字を挿入する処理は，第3.3節で説明したように，現在位置から右側を1文字分右に転送して空きを作り，そこに文字を挿入する．その空きを作る処理は，次のようにループを作って行う．

```
p = buf + strlen(buf); ———— pは文字列の最後（\0）を指す
while (p >= (buf + col)) {
    *(p + 1) = *p;
    p--;
}
```

文字列の転送を文字列の後ろから実行していることに注意．前から転送すると，途中でも，もとの文字列が上書きされてしまうからである．

転送が終わったら挿入位置colに文字eを挿入する．と同時に，colを1文字右（挿入した文字の右）に移動する．

```
buf[col++] = e;
```

これで，ASCII文字の挿入は終わりである．

8.6◆漢字の挿入

漢字の場合は，バッファに2バイト分の領域が必要であるということを考慮する必要がある．

```
if (strlen(buf) >= width - 1) { ——— 2バイト分の空きがあるかチェック
    beep(); ———— 漢字を挿入する余裕がない
    return;
}
else {
    // 後ろからコピーして2文字分の空き領域を作る
    p = buf + strlen(buf);  // pは\0を指す
    while (p >= (buf + col)) {
        *(p + 2) = *p;
        p--;
    }
    e1 = (e >> 8);
    e2 = (e & 0xff);
```

```
        buf[col++] = e1;
        buf[col++] = e2;
    }
```

以上で，1文字を挿入する関数inscharが完成する．次に，そのリストを示す．

```
void EditBox::inschar(int e)
{
    char *p;
    int e1, e2;

    if (e > 0x7f) {  // 漢字の場合
        if (strlen(buf) >= width - 1) {
            beep();
            return;
        }
        else {
            // 後ろからコピーして空き領域を作る
            p = buf + strlen(buf);  // pは\0を指す
            while (p >= (buf + col)) {
                *(p + 2) = *p;
                p--;
            }
            e1 = (e >> 8);
            e2 = (e & 0xff);
            buf[col++] = e1;
            buf[col++] = e2;
        }
    }
    else {          // ASCIIの場合
        if (strlen(buf) >= width) {
            beep();
            return;
        }
        else {
            // 後ろからコピーして空き領域を作る
            p = buf + strlen(buf);  // pは\0を指す
            while (p >= (buf + col)) {
                *(p + 1) = *p;
                p--;
            }
            buf[col++] = e;
        }
    }
    move(home_y, home_x);
    addstr(buf);
    move(home_y, home_x + col);
}
```

8.7◆文字の削除

文字の削除は，次の2つの下請け関数を使っている．

```
bschar ―――――――――― カーソルの左の文字を削除する
delchar ――――――――― カーソル上の文字を削除する
```

このうち本質的なのは，delcharの方である．なぜなら，bscharは内部でdelcharを呼び出して処理を委託しているからである．カーソルを1文字左に移動してdelcharすれば，bscharと同じことになる．

```
void EditBox::bschar()
{
    if (col <= 0) ―――――――― バッファの左端にいるときは削除できない
        beep();
    else {
        moveleft(); ―――――― カーソルを左に移動して
        delchar(); ――――――― カーソル上の文字を削除する
    }
}
```

delcharの処理も複雑ではあるが，例外を1つずつ片づけているに過ぎない．つまり，次のようなことを1つずつチェックしながら場合分けをする．

- カーソルが右端にあるときは削除できない
- 削除する文字が漢字かどうか

次にdelcharのリストを示す．

```
void EditBox::delchar()
{
    char *p;

    if (col >= strlen(buf)) { ―――――― バッファの右端にいるときは削除できない
        beep();
        return;
    }
    if (buf[col] & 0x80) { ――――――― 漢字の場合
        p = buf + col + 2;
        while (*p) {
            *(p - 2) = *p;
```

```
            p++;
        }
        *(p - 2) = '\0';
    }
    else {  ――――――――――――――― ASCII文字の場合
        p = buf + col + 1;
        while (*p) {
            *(p - 1) = *p;
            p++;
        }
        *(p - 1) = '\0';
    }
    cls();  ――――――――――――――― ひらがなを正しく消すための処理
    move(home_y, home_x);
    addstr(buf);
    move(home_y, home_x + col);
}
```

9 全リスト

EditBoxの全リストを示す．このリストは，第6節で説明したmainを組み込んであるので，そのままコンパイルして実行することができる．

```
// editbox.C -- ラベル付きの1行スクリーンエディタ
//                  by K. Tsukagoshi
//                  1998.7.16 - 1998.10.5

#include <curses.h>

const int CTRL_B = 0x02;
const int CTRL_D = 0x04;
const int CTRL_F = 0x06;
const int CTRL_H = 0x08;
const int CTRL_Q = 0x11;
const int CTRL_U = 0x15;

// エディットボックスクラス
// 編集バッファ
//      先頭アドレス          : buf
//      入力可能文字数        : width
```

```
//        バッファのサイズ        : width + 1 ('\0'が必要)
//        バッファ上の入力位置    : col
//        画面上の座標            : home_x + col
class EditBox {
    int home_x;                                 // ホーム座標
    int home_y;

    char label[20];                             // 編集領域の左に表示するラベル

    char *buf;                                  // 編集バッファ
    int col;                                    // buf上の次の入力位置
    int label_x;                                // ラベルの先頭
                                                // 編集領域の先頭はhome_x
    int width;                                  // 最大入力可能文字数

    void cls();
    void empty();
    void inschar(int e);
    void bschar();
    void delchar();
    void moveleft();
    void moveright();
public:
    EditBox(int x, int y, char *l, int w);
    ~EditBox();
    void show();
    void setfocus();
    char *get_text();
    void handle_event(int e);
};

// ラベルと編集領域の初期化
//        (x, y) :ラベルの位置
//        l      :ラベルに表示する文字列
//        w      :編集領域の文字数
EditBox::EditBox(int x, int y, char *l, int w)
{
    // ラベルの初期化
    label_x = x;
    strcpy(label, l);

    // 編集領域の初期化
    home_x = x + strlen(l) + 2;
    home_y = y;
```

```
    width = w;
    col = 0;

    buf = new char[width + 1];
    *buf = '\0';
}

EditBox::~EditBox()
{
    delete [] buf;
}

// 編集領域をスペースでクリアする
// バッファには手をつけない
void EditBox::cls()
{
    attrset(WA_UNDERLINE);
    move(home_y, home_x);
    for (int i = 0; i < width; i++)
        addch(' ');
    refresh();
}

// 編集領域を表示領域，バッファともに空にする
void EditBox::empty()
{
    cls();
    *buf = '\0';
    col = 0;
    move(home_y, home_x);
}

// エディットボックスを表示する
void EditBox::show()
{
    attrset(WA_NORMAL);
    move(home_y, label_x);
    addstr(label);
    attrset(WA_UNDERLINE);
    cls();
}

// フォーカスを得たときの処理
void EditBox::setfocus()
```

```
{
    move(home_y, home_x);
    attrset(WA_UNDERLINE);
    refresh();
}

// 現在のテキストを返す
char *EditBox::get_text()
{
    return buf;
}

// 1文字挿入
void EditBox::inschar(int e)
{
    char *p;
    int e1, e2;
    if (e > 0x7f) {  // 漢字の場合
        if (strlen(buf) >= width - 1) {
            beep();
            return;
        }
        else {
            // 後ろからコピーして空き領域を作る
            p = buf + strlen(buf);  // pは\0を指す
            while (p >= (buf + col)) {
                *(p + 2) = *p;
                p--;
            }
            e1 = (e >> 8);
            e2 = (e & 0xff);
            buf[col++] = e1;
            buf[col++] = e2;
        }
    }
    else {                              // ASCIIの場合
        if (strlen(buf) >= width) {
            beep();
            return;
        }
        else {
            // 後ろからコピーして空き領域を作る
            p = buf + strlen(buf);  // pは\0を指す
```

```
            while (p >= (buf + col)) {
                *(p + 1) = *p;
                p--;
            }
            buf[col++] = e;
        }
    }
    move(home_y, home_x);
    addstr(buf);
    move(home_y, home_x + col);
}

// カーソル上の1文字を削除
void EditBox::delchar()
{
    char *p;

    if (col >= strlen(buf)) {    // バッファの右端にいるときは削除できない
        beep();
        return;
    }
    if (buf[col] & 0x80) {       // 漢字
        p = buf + col + 2;
        while (*p) {
            *(p - 2) = *p;
            p++;
        }
        *(p - 2) = '\0';
    }
    else { // ASCII
        p = buf + col + 1;
        while (*p) {
            *(p - 1) = *p;
            p++;
        }
        *(p - 1) = '\0';
    }
    cls(); // ひらがなを正しく消すには一度refreshする
    move(home_y, home_x);
    addstr(buf);
    move(home_y, home_x + col);
}

// 左の1文字を削除
```

```
void EditBox::bschar()
{
    if (col <= 0)                          // バッファの左端にいるときは削除できない
        beep();
    else {
        moveleft();
        delchar();
    }
}

// 左に移動
void EditBox::moveleft()
{
    if (col <= 0)                          // バッファの左端にいるときは移動できない
        beep();
    else if (buf[--col] & 0x80) {  // 漢字
        --col;
        move(home_y, home_x + col);
    }
    else  // ASCII
        move(home_y, home_x + col);
}

// 右に移動
void EditBox::moveright()
{
    if (!buf[col])                         // 最後の文字の右にいるときは移動できない
        beep();
    else if (buf[col++] & 0x80) {  // 漢字
        col++;
        move(home_y, home_x + col);
    }
    else  // ASCII
        move(home_y, home_x + col);
}

// エディットボックスのイベントハンドラ
void EditBox::handle_event(int e)
{
    if (e == CTRL_B)
        moveleft();
    else if (e == CTRL_F)
        moveright();
    else if (e == CTRL_H)                  // 左の文字削除
```

```
        bschar();
    else if (e == CTRL_D)               // カーソル上の文字削除
        delchar();
    else if (e == CTRL_U)               // 編集領域を空にする
        empty();
    else if (e < 0x20)                  // その他のコントロールコードは無効
        beep();
    else                                // 1文字挿入追加
        inschar(e);
    refresh();
}

// キー入力を行う
// 漢字の場合は2バイトにして返す
// 上位バイトが0ならASCIIコード
// 半角2バイト文字には非対応
int getevent()
{
    int c, c2;

    if ((c = getchar()) > 0x7f) {
        c2 = getchar();
        return (c << 8) | c2;
    }
    else
        return c;
}

int main()
{
    int e;

    initscr();                          // cursesの使用開始
    noecho();                           // キー入力時に表示しない
    raw();                              // rawモードにする

    EditBox eb(10, 5, "文字列編集:", 10);
    eb.show();
    eb.setfocus();

    // イベントの検出と配送
    while ((e = getevent()) != CTRL_Q)
        eb.handle_event(e);
```

```
    // 入力結果の表示
    move(7, 10);
    attrset(WA_NORMAL);
    addstr("編集結果    :  ");
    attrset(WA_UNDERLINE);
    addstr(eb.get_text());
    move(11, 10);
    attrset(WA_REVERSE);
    addstr("何かキーを押すと終了");

    refresh();
    getchar();

    endwin();

    return 0;
}
```

第34章 イベント駆動型アプリケーションフレームワーク

終章として，イベント駆動型の**アプリケーションフレームワーク**を作成する．フレームワーク（*framework*）というのは枠組のことで，アプリケーションフレームワークは，アプリケーションそのものをカプセル化したライブラリのことである．

イベント駆動型のアプリケーションは，C++と相性がよい．アプリケーションフレームワークは，まさにC++の終章を飾るのに相応しいテーマといえる．

1 アプリケーションフレームワーク

再利用可能なプログラムの部品を**ライブラリ**という．

ライブラリというと，通常はキーの入力ルーチンとか，メニューとか，エディタなどといった「プログラムの一部分」をカプセル化したものが一般的である．しかし，この章で取り上げる**アプリケーションフレームワーク**は，プログラム全体——すなわち，アプリケーションそのものをカプセル化したものである．

アプリケーションフレームワークを利用することにより，次のようなメリットを得ることができる．

- cursesライブラリの初期化と後処理（←cursesライブラリを使う場合）といった，どのアプリケーションにも必要な処理を毎回プログラムしなくて済む
- アプリケーションの見ばえや使い勝手が統一される

一度好みのアプリケーションフレームワークを作っておけば，細々(こまごま)した雑用に煩わされることなく，それぞれのアプリケーションに専念することができる．

2 オブジェクトベースのライブラリ

本書は，C++の入門書である．したがって，この章で作成するアプリケーションフレームワークも，C++の特色を活かした**オブジェクトベース**のものになる．

ライブラリには**関数ベース**のものと**オブジェクトベース**のものがある．両者の違いは，そのライブラリが「関数で作られている」か「オブジェクトで作られている」かにある．「オブジェクトは関数の集合である」と見ることもできるので，オブジェクトベースのライブラリは，次のような特色を持つ．

- **手順の隠蔽**
- **再コンパイルすることなくライブラリを拡張できる**

以下，アプリケーションフレームワークを例に，オブジェクトベースのライブラリについて説明していく．

2.1◆手順の隠蔽

「手順の隠蔽」を実感するには，アプリケーションフレームワークを使ってみるのがよい．

本章のアプリケーションフレームワークは，次の2つのファイルで出来上がっている．

```
app.h
app.C
```

アプリケーションフレームワークを利用するアプリケーションは，まずapp.hをインクルードする．

```
#include "app.h"
```

アプリケーションフレームワークは，**App**というクラスでできている．そこで，

ユーザーアプリケーションを生成するには，Appのオブジェクトを作る．

App app; ——— appはApp型のオブジェクト

ただし，Appのコンストラクタは3つの引数をとる．

ヘッダ ——— アプリケーションの上部に表示する文字列
フッタ ——— アプリケーションの下部に表示する文字列
使用するウィジェットの数

したがって，Appオブジェクトを作るとき，これらの引数を指定する必要がある．

ウィジェットというのは，編集ボックスのようなプログラムの部品である．ここでは，最低限のアプリケーションを作るため，次の3つの引数を指定することにした．

ヘッダ ——— “テストアプリケーション”
フッタ ——— “^Qで終了します”
ウィジェットの数 ——— 0

というわけで，Appオブジェクトの作成は次のようになる．

App app("テストアプリケーション", "^Qで終了します", 0);

Appオブジェクトを作成すれば，あとはメンバー関数のrunを呼び出すだけで，アプリケーションを実行することができる．

app.run();

たった，これだけである．これだけでcursesを利用したプログラムが動いてしまう．

以上をまとめると，アプリケーションフレームワークを使った最低限のアプリケーションを作るには，次のようなメインプログラムを書けばよい．

```
// go.C -- 裸のアプリケーションフレームワークを使用する
//                          by K. Tsukagoshi
//                          1998.10.25

#include "app.h"

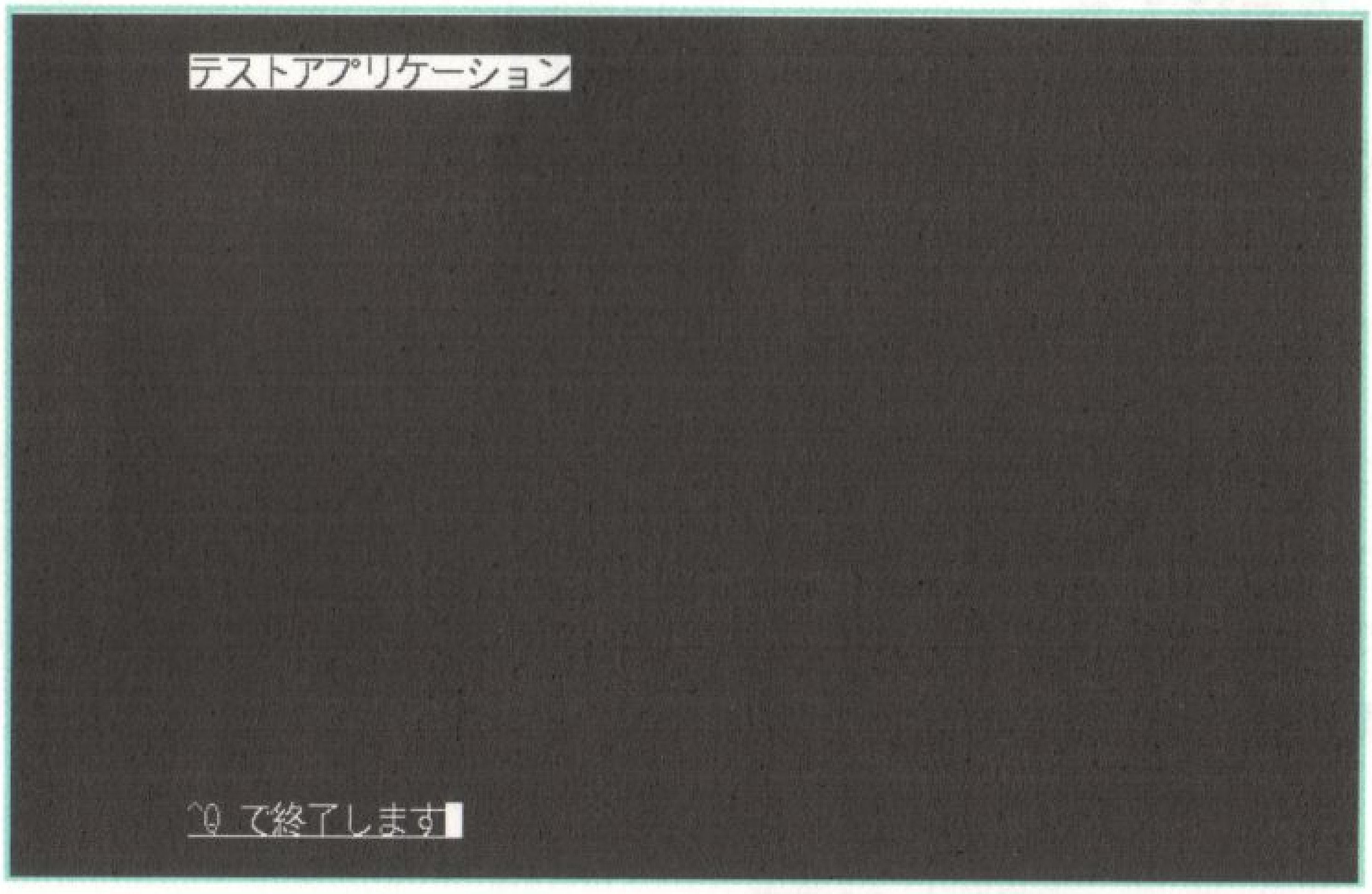
int main()
{
    App app("テストアプリケーション", "^Qで終了します", 0);
```

```
    app.run();

    return 0;
}
```

次に，このプログラムを実行したところを示す．

図34-1▶



画面の上部には，タイトルの“テストアプリケーション”の文字が反転表示されている（cursesライブラリが動いていることがわかる）．また，画面の下部にはフッタの“^Qで終了します”の文字が表示されている．

プログラム実行中は，アプリケーションフレームワークに組み込まれた「イベントの検出ルーチン」が，ユーザーのキー入力を監視している．そして，キーが入力されると，適切なウィジェット（この例ではウィジェットを組み込んでいないが）に配送される．

また，このアプリケーションフレームワークには，**ホットキー**を処理するルーチンが組み込まれている．Ctrlキーを併用したキーに特別な意味を持たせることができる．デフォルトのホットキーは，Ctrl＋Qである．Ctrl＋Qを押すと，プログラムが終了する．

以上が，アプリケーションフレームワークのデフォルトの動作である．cursesライブラリが使われている．裏でたくさんのことが行われているのが予想される．

もし，アプリケーションフレームワークを使わないで同等のプログラムを開発しようとしたら，かなりたくさんの関数を書かなければならない．また，それらの関数を使用する手順も重要になる．

cursesを初期化

↓

アプリケーションの初期化をする

↓

画面を描く

↓

キー入力を監視する

↓

Ctrl＋Qが押されたら後処理をする

↓

cursesの後処理をする

これらに関わる関数を順序正しく呼び出す必要がある．しかし，このメインプログラムでやったことは，Appオブジェクトを作って，そのrunメンバー関数を呼び出しただけである．このようにオブジェクトベースのライブラリでは，**複雑な関数の実行手順を隠蔽する**ことができる．

2.2◆再コンパイル不要の拡張

オブジェクトベースのライブラリにおける2番目の特色は，「ライブラリを拡張するのに再コンパイルが要らない」ということである．

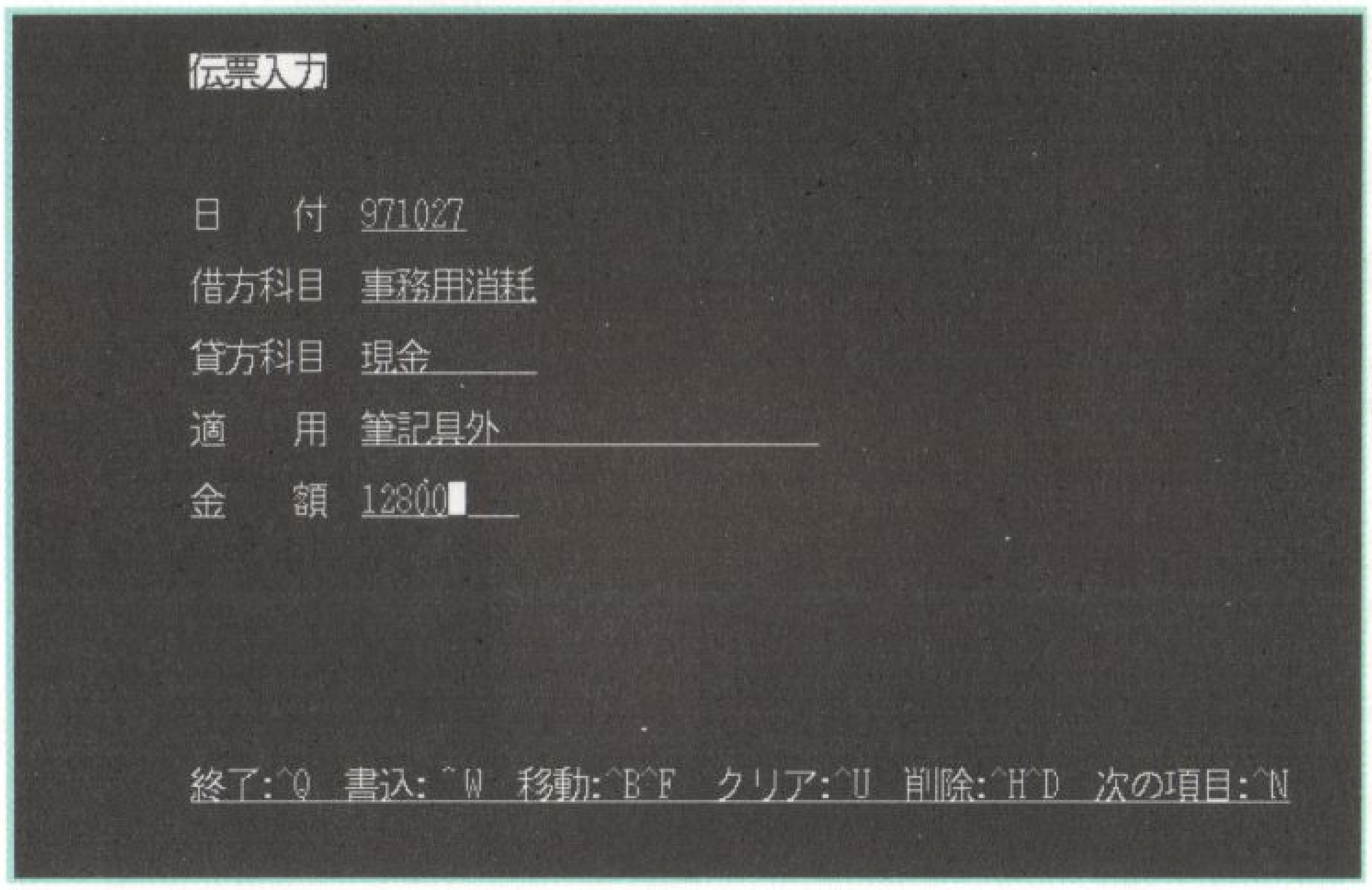
関数ベースのライブラリでは，機能を拡張するとき，もとのソースを引っぱり出してきて，そのソースに新しいコードを追加していくという方法をとる．つまり，ソースファイルを書き換えているので，ライブラリを再コンパイルする必要がある．

しかし，オブジェクトベースのライブラリでは，（極端にいえば）もとのソースファイルを必要としない（ヘッダファイルは必要）．コンパイル済みのオブジェクトを使って，そのライブラリを拡張することができる．

これもアプリケーションフレームワークを例にするとわかりやすい．実際，アプリケーションフレームワークに機能を追加するときは，この方法を使う．

例として，アプリケーションフレームワークを使って，次のようなアプリケーションを作成してみる（具体的な作成方法は第3節で説明する）．

図34-2▶



この図と第1節の**図34-1**を比較されたい．前に示したものが，デフォルトのAppクラスの機能を使ったものである．それに対して，今度のアプリケーションには，たくさんの編集ボックスが追加されているのがわかる．

このようなアプリケーションを作成するには，Appクラスを拡張することによって実現する．といっても，Appクラスを書き換えるのではない．Appクラスを継承した新しいクラスを作り，そこに拡張機能を書いていくのである．

たとえば新しいアプリケーションの拡張クラスの名前をExtAppとすれば，次のようなクラスを作る．

```
class ExtApp : public App {
    ............
};
```

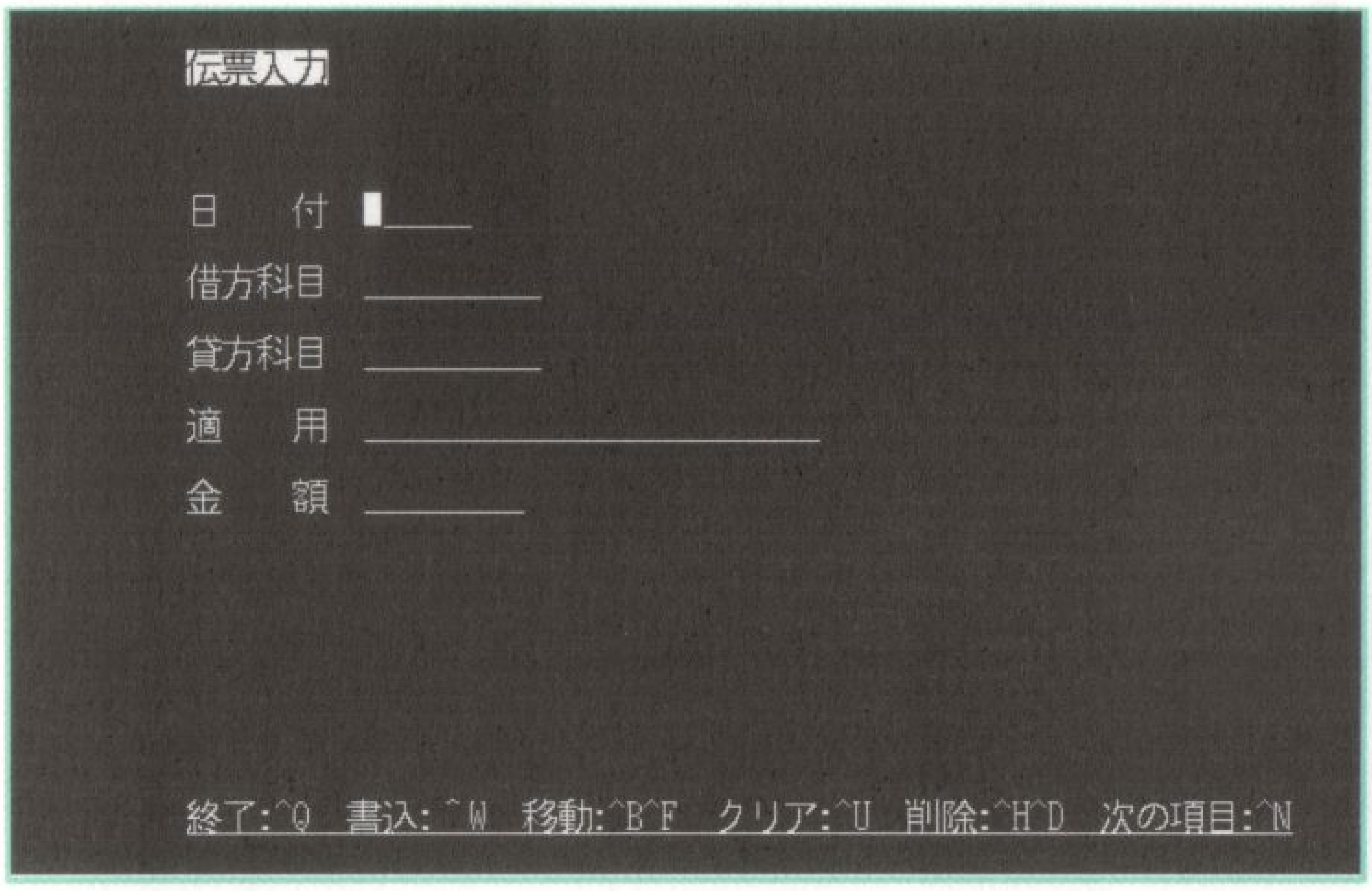
ExtAppは，アプリケーションクラスAppを継承しているので，Appのすべての機能を持っている．さらに，ここに新しい定義を書き加えることにより，アプリケーションの機能を拡張することができる．

具体的な拡張については，次の第3節で説明する．

3 サンプルアプリケーション

この節では，Appクラスを拡張して新しいアプリケーションを作る方法を説明する．

図34-3▶



これは，伝票を入力するためのインターフェースを提供するものである．PerlやAwkなどのスクリプト言語でデータ処理をするときなどのフロントエンドに使える．

各項目を入力するのに**編集ボックス**を利用しているのでスクリーンエディットが可能である．もちろん，アプリケーションフレームワークには，第33章で説明した編集ボックスが組み込まれているので，編集ボックスのためのコードを追加する必要はない．

3.1◆コンストラクタの拡張

まず次が，オリジナルのアプリケーションクラスAppを継承したExtAppクラスである．

```
class ExtApp : public App {
        ............
};
```

もとのAppは**デフォルトコンストラクタ**（引数なしのコンストラクタ）を持っていない．そこで，ExtAppにも引数付きのコンストラクタを定義して，引数をAppのコンストラクタに渡す必要がある．

```
class ExtApp : public App {
        ............
public:
        ExtApp(char *h, char *f, int n);
};
```

```
ExtApp::ExtApp(char *h, char *f, int n) : App(h, f, n)
{
        ............
};
```

拡張アプリケーションの初期化を行うコードは，この中に書く．

3.2◆データファイル

このアプリケーションは，データファイルを使用する．簡単のため，ファイル名は“data”に固定した．入力した伝票のデータを，このファイルにセーブする．そこで，ファイルをオープンする処理をExtAppのコンストラクタに書く．

```
ExtApp::ExtApp(char *h, char *f, int n) : App(h, f, n)
{
        if ((fp = fopen("data", "a")) == NULL) {
                endwin();
                fputs("can't open \"data\"\n", stderr);
                exit(1);
        }
}
```

fpは，オープンしたファイルへのファイルポインタである．この変数をprivateメンバーとして用意する．

```
class ExtApp : public App {
        FILE *fp;
public:
        ExtApp(char *h, char *f, int n);
};
```

一方，デストラクタを用意してその中でオープンしたデータファイルをクローズする．

```
class ExtApp : public App {
        FILE *fp;
public:
```

```
        ExtApp(char *h, char *f, int n);
        ~ExtApp(); ―――――――― デストラクタ
};

ExtApp::~ExtApp()
{
        if (fp != NULL)
            fclose(fp);
}
```

3.3◆ウィジェットの挿入

アプリケーションには，編集ボックスなどのウィジェットを追加して使うことができる．

ウィジェットをアプリケーションに挿入するには，メンバー関数のadd_widgetを使う．add_widgetは，すでにAppの中で定義されているので，ExtAppで再定義する必要はない．

```
ExtApp app(……);
app.add_widget(ウィジェットへのポインタ);
```

add_widgetには，挿入するウィジェットへのポインタを指定する．たとえば編集ボックスを挿入するなら，次のようにする．

```
app.add_widget(new EditBox(INDENT,　5, "日　　付", 6));
```

newを使って引数の中で編集ボックスオブジェクトを作っていることに注意．なお，編集ボックスオブジェクトの廃棄は無視してよい．ちゃんとAppのデストラクタが，挿入されたウィジェットのメモリを自動的に解放してくれるようになっている．

3.4◆ホットキー

アプリケーションに**ホットキー**を組み込むこともできる．たとえば，この節のサンプルプログラムでは[Ctrl]＋[W]というホットキーが定義されていて，このキーが押されると，入力中のデータをファイルに書き込むようになっている．

ホットキーを定義するには，Appクラスで定義されているhotkeyという仮想関数をExtAppでオーバーロードする．

```
class ExtApp : public App {
    FILE *fp;
public:
    ExtApp(char *h, char *f, int n);
    ~ExtApp();
    int hotkey(int e); ―――――― オーバーロードする
};
```

hotkeyの引数eには，入力されたキーのコードが入っている．したがって，eが Ctrl＋W なら，データをファイルに書き込む処理を書けばよい．

```
int ExtApp::hotkey(int e)
{
    if (e == CTRL_W)
        // データをファイルに書き込む
}
```

hotkeyはint型のデータを返す．その値は，与えられたイベントeをhotkeyの中で処理したかを示すものである．

```
0 ―――――――――― イベントeを処理していない
1 ―――――――――― イベントeを処理した
```

たとえば，この例では，入力されたキーが Ctrl＋W の場合はhotkeyの中で処理を済ませるが，その他のキーの場合は何もしない．したがって，hotkeyの関数値は次のようになる．

```
int ExtApp::hotkey(int e)
{
    if (e == CTRL_W) {
        // データをファイルに書き込む
        return 1;
    }
    return 0;
}
```

この関数値は，イベントの配送ルーチンで使われる．すなわち，hotkey関数を呼び出し，もしそのイベントがまだ処理されていないなら，次の適切なオブジェクトに，そのイベントを配送する．

3.5◆ウィジェットの配列

アプリケーションに挿入されたウィジェットは，挿入順にwidgetという配列の要素になっている．

widgets[0]――1番目に挿入されたウィジェットへのポインタ
widgets[1]――2番目に挿入されたウィジェットへのポインタ
widgets[2]――3番目に挿入されたウィジェットへのポインタ
widgets[3]――4番目に挿入されたウィジェットへのポインタ
....

したがって，配列widgetsを参照することにより，各ウィジェットのメンバー関数を呼び出すことができる．この節のサンプルアプリケーションでは，5つの編集ボックスを使っている．各編集ボックスは，次のように挿入されている．

widgets[0]――“日付”項目のための編集ボックス
widgets[1]――“借方科目”項目のための編集ボックス
widgets[2]――“貸方科目”項目のための編集ボックス
widgets[3]――“適用”項目のための編集ボックス
widgets[4]――“金額”項目のための編集ボックス

widgetの要素の型は，実はEditBox *ではない．Widget *である．もしこれをEditBox *にしてしまうと，ウィジェットとして編集ボックスしか使えなくなってしまう．

ボタンとかリストボックスとかといった，いろいろなウィジェットを使いたいなら，それらに共通の仮想クラスを作って，具体的なウィジェットはそこから派生するようにする．その仮想クラスが**Widget**である．こうすれば，すべてのウィジェットをWidgetで統一的に管理することができる．

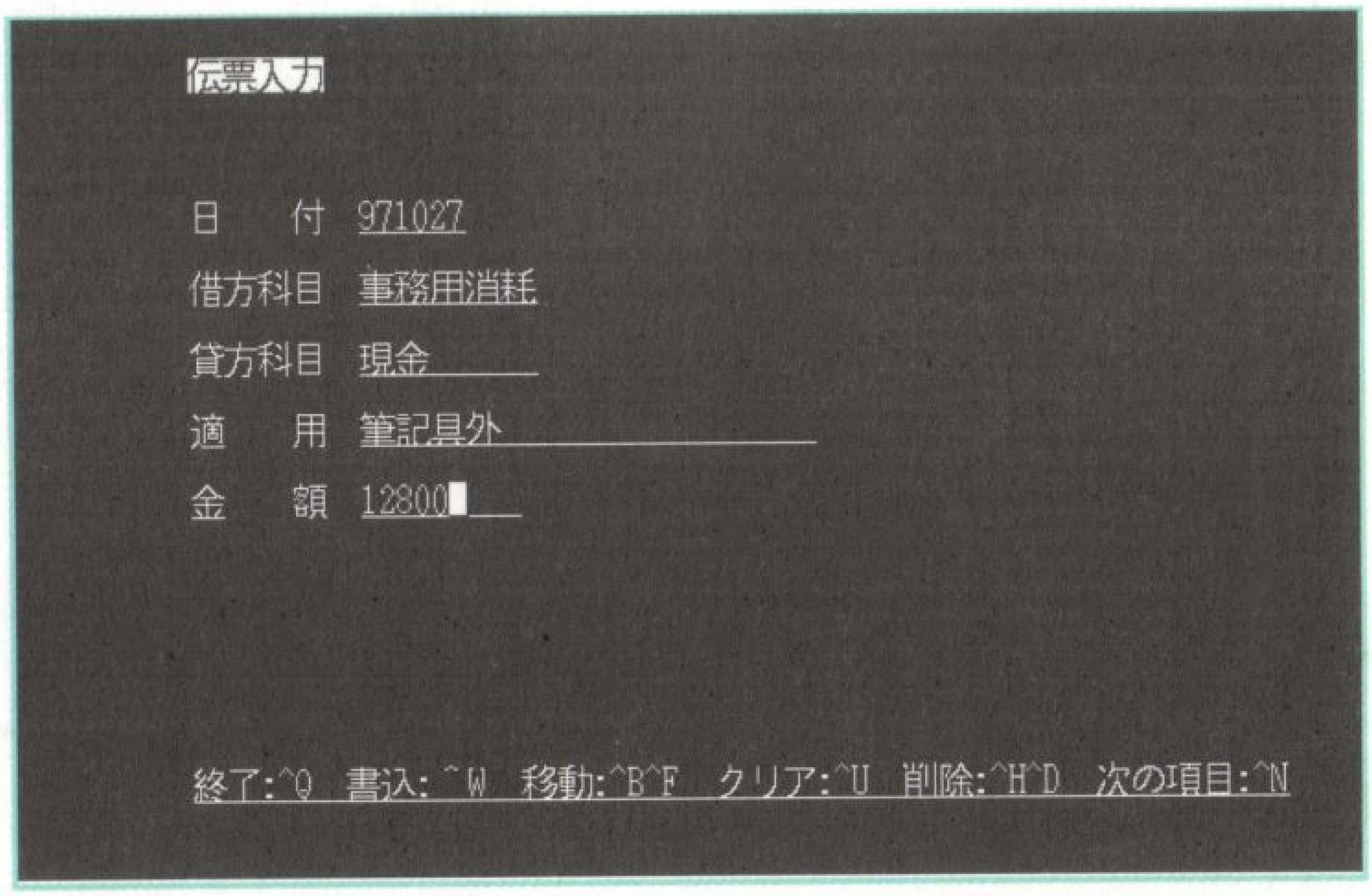
EditBoxもWidgetの派生クラスとして実装されている．したがって，widgetsの要素をEditBoxとして扱いたいなら，(EditBox *)で型キャストする必要がある．たとえば次は“日付”項目に入力されている文字列を参照するものである．

```
(EditBox *)widgets[0])->get_text()
```

3.6◆データの保存

このアプリケーションでは，Ctrl＋Wが押されると現在入力中のデータをdataという名前のファイルに追加する．たとえば，現在，次のようなデータが入力されているとする．

図34-4▶



この状態でCtrl＋Wを入力すると，dataの内容は次のようになる．

図34-5▼

```
% cat data
971027:事務用消耗:現金:筆記具外:12800
% _
```

このように各項目はコロンで区切られる．データを入力するたびに，この後ろにデータが追加されていく．

ホットキーのCtrl＋Wにこのような機能を持たせるためには，hotkeyに次のようなコードを書けばよい．

```
int ExtApp::hotkey(int e)
{
    if (e == CTRL_W) {
        fprintf(fp, "%s:%s:%s:%s:%s\n",
            ((EditBox *)widgets[0])->get_text(),
            ((EditBox *)widgets[1])->get_text(),
```

```
                ((EditBox *)widgets[2])->get_text(),
                ((EditBox *)widgets[3])->get_text(),
                ((EditBox *)widgets[4])->get_text());
            beep(); ──────────────────────── ベルを鳴らす
            refresh(); ────────────────────── リフレッシュが必要
            return 1;
        }
        else
            return 0;
}
```

このプログラムでは，fprintfでデータを書き込んだ後でベルを鳴らしている．beepはcursesの関数である．beepも他の描画用の関数と同じように，refreshを呼ばないと実行されない．

3.7◆リスト

次に，この節で説明してきた拡張アプリケーションのリストの全体像を示す．

```
// in.C --振替伝票入力インターフェース
//                                                  by K. Tsukagoshi
//                                                  1998.10.27

#include <curses.h>
#include "app.h"

// 固有のアプリケーションクラス
class ExtApp : public App {
    FILE *fp;
public:
    ExtApp(char *h, char *f, int n);
    ~ExtApp();
    int hotkey(int e);
};

// データファイルのオープン
// ファイル名は"data"に固定
ExtApp::ExtApp(char *h, char *f, int n) : App(h, f, n)
{
    if ((fp = fopen("data", "a")) == NULL) {
        endwin();
        fputs("can't open \"data\"\n", stderr);
```

```
            exit(1);
        }
    }

    ExtApp::~ExtApp()
    {
        if (fp != NULL)
            fclose(fp);
    }

    // ホットキーの処理
    // ^Wでカレントレコードをファイルに書き込む
    // 処理済みは1，未処理は0を返す
    int ExtApp::hotkey(int e)
    {
        if (e == CTRL_W) {
            fprintf(fp, "%s:%s:%s:%s:%s\n",
                ((EditBox *)widgets[0])->get_text(),
                ((EditBox *)widgets[1])->get_text(),
                ((EditBox *)widgets[2])->get_text(),
                ((EditBox *)widgets[3])->get_text(),
                ((EditBox *)widgets[4])->get_text());
            beep();
            refresh();      // for beep
            return 1;
        }
        else
            return 0;
    }

    int main()
    {
        ExtApp app(
            "伝票入力",
            "終了:^Q  書込:^W  移動:^B^F  クリア:^U  削除:^H^D  次の項目:^N",
            5
        );

        app.add_widget(new EditBox(INDENT,  5, "日    付", 6));
        app.add_widget(new EditBox(INDENT,  7, "借方科目", 10));
        app.add_widget(new EditBox(INDENT,  9, "貸方科目", 10));
        app.add_widget(new EditBox(INDENT, 11, "適    用", 26));
        app.add_widget(new EditBox(INDENT, 13, "金    額", 9));
```

```
    app.run();

    return 0;
}
```

4 cursesの管理

この節からAppクラスそのものの説明に入る．まず，この節でcursesライブラリの初期化と後処理をまとめる．

4.1◆cursesの制御

Appクラスは，**cursesライブラリ**を使って画面の制御を行っている．cursesを使うためには，あらかじめ初期化が必要である．それをinit_cursesという専用の関数の中で行っている．

init_cursesは，privateメンバーである．cursesの初期化のような決まりきった処理は，ユーザーに見せる必要はないからである．

```
class App {
private:
    void init_curses(); ―――――― privateメンバーにする
    ............
pubic:
    ............
};

void App::init_curses()
{
    initscr(); ―――――――――――――― cursesの使用開始
    start_color(); ――――――――――― 色の使用を可能にする
    raw(); ―――――――――――――――――― rawモードにする
}
```

init_cursesは，Appのコンストラクタの中から呼び出される．こうすれば，cursesの初期化を自動化することができる．

```
// Appのコンストラクタ
App::App(char *h, char *f, int n)
{
    init_curses(); ――――― 何はともあれcursesを初期化
    …………
}
```

同様に，cursesの後処理をAppのデストラクタの中に置く．

```
App::~App()
{
    …………
    endwin(); ――――― cursesの後処理
}
```

これでcursesの初期化と後処理に煩わされることなく，cursesの使用に専念することができる．

4.2◆ヘッダとフッタ

cursesが利用できるようになると，ヘッダとフッタを表示することができる．

ヘッダ ――――― 画面上部に表示する文字列
フッタ ――――― 画面下部に表示する文字列

この表示をAppのコンストラクタの中で行う．ヘッダとフッタのための文字列は，コンストラクタの引数として渡される．

```
App::App(char *h, char *f, int n)
{
    init_curses(); ――――― cursesを初期化した後

    move(1, INDENT); ――――― ヘッダの表示
    attrset(WA_REVERSE);
    addstr(h);
```

```
        move(LINES - 2, INDENT);  ―――― フッタの表示
        attrset(WA_UNDERLINE);
        addstr(f);

        attrset(WA_NORMAL);
        refresh();  ―――――――――――― ここで表示される
        ............
    }
```

5 ウィジェットの管理

アプリケーションには，**ウィジェット**（*widget*）と呼ばれるプログラムの部品を貼り付けて利用することができる．この章で扱うアプリケーションフレームワークで使えるウィジェットは，**編集ボックス**だけである．編集ボックスのコードは，第33章で説明したものを，ほぼそのまま利用している．

編集ボックス以外のウィジェットも追加できるようにするためには，仮想的な基本クラスを作って，すべてのウィジェットをその派生クラスにするのがよい．

5.1◆ウィジェットの基本クラス

次が，すべてのウィジェットの基本クラスとなるWidgetである．

```
class Widget {
        ............
};
```

クラスWidgetは，すべての部品に共通する性質や動作を記述する．

性質	
ウィジェットの位置	(home_x, home_y)
動作	
ウィジェットの表示	show()
ウィジェットにフォーカスを当てる	setfocus()
イベントの処理	handle_event()

したがって，ウィジェットのclass宣言は次のようになる．

```
class Widget {
protected:
    int home_x;
    int home_y;
public:
    Widget(int x, int y);
    virtual ~Widget() {};
    virtual void show() = 0;
    virtual void setfocus() = 0;
    virtual void handle_event(int e) = 0;
};
```

メンバー変数のhome_x，home_yは，その部品の左上の座標を記憶する．

メンバー関数のshow，setfocus，handle_eventが純粋仮想関数になっていることに注意．これは，部品によって，その動作が異なるからである．たとえばshowは，その部品を表示する．しかし，部品によって表示のしかたが異なる．したがって，showは部品ごとにオーバーロードして，その部品に相応しい描き方をする必要がある．

5.2◆編集ボックス

ウィジェットの例として**編集ボックス**を組み込んである．編集ボックスもウィジェットの1つなので，Widgetクラスの派生クラスにする．

```
class EditBox : public Widget {
    ............
};
```

第33章で作成した編集ボックスは，単独のクラスであった．

```
class EditBox {  ―――― 派生していない
    ............
};
```

しかし，いまや編集ボックスはWidgetクラスの派生クラスに組み込まれている．したがって，EditBoxクラスが持っていた一部の属性は，Widgetクラスに回されて

いる．新しいEditBoxクラスの宣言は次のようになる．

```
class EditBox : public Widget {
    char label[20];         // 編集領域の左に表示するラベル
    int width;              // 最大入力可能文字数
    int label_x;            // ラベルの先頭
                            // 編集領域の先頭はhome_x
    char *buf;              // 編集バッファ
    int col;                // buf上の次の入力位置

    void cls();
    void empty();
    void inschar(int e);
    void bschar();
    void delchar();
    void moveleft();
    void moveright();

public:
    EditBox(int x, int y, char *l, int w);
    ~EditBox();
    void show();
    void setfocus();
    char *get_text();
    void handle_event(int e);
};
```

その他の具体的なコードは，第33章で説明したものとほぼ同じである．

5.3◆WidgetとApp

WidgetとAppは別々のクラスである．

Widget――ウィジェットをカプセル化したクラス
App――アプリケーションをカプセル化したクラス

ウィジェットはAppクラスによって管理される．そのとき次の2つのメンバー変

数が使用される．

nwidgets ──── 使用するウィジェットの最大数
widgets ──── ウィジェットの配列
active ──── 現在アクティブなウィジェット
next ──── 現在挿入されているウィジェットの数
初期値はコンストラクタでゼロに設定

widgets[active] が，現在アクティブな（イベントを受け取る）ウィジェットを指す．また，widgets[next] に新しいウィジェットを挿入することができる．

これらの変数は，Appクラスのコンストラクタで初期化される．

```
App::App(char *h, char *f, int n)
{
    …………
    nwidgets = n;
    widgets = new (Widget *)[nwidgets];
    active = next = 0;
}
```

この結果，動的配列widgetsが指定された数の要素数を持つようにメモリが確保される．

5.4◆ウィジェットの挿入

ウィジェットをアプリケーションに挿入するためのメンバー関数がadd_widgetである．次にadd_widgetのリストを示す．

```
void App::add_widget(Widget *pw)
{
    widgets[next++] = pw;
}
```

pwがウィジェットへのポインタである．それを配列の要素に代入し，次の要素を挿入できるようにnextをインクリメントしている．

アプリケーションに挿入されている全ウィジェットは，showメンバー関数を実行することで表示される．

```
void App::show()
{
    for (int i = 0; i < next; i++) ──── 全ウィジェットを表示し
        widgets[i]->show();
    if (next > 0) ──── アクティブなウィジェットに
        widgets[active]->setfocus(); ──── フォーカスを当てる
}
```

5.5◆ウィジェットの廃棄

Appクラスに挿入された全ウィジェットは，Appクラスのデストラクタによって自動的に廃棄される．そのとき，ウィジェットのために割り当てられたメモリも解放される．

```
App::~App()
{
    for (int i = 0; i < next; i++) ──── ウィジェットのメモリを解放
        delete widgets[i];
    delete [] widgets; ──── 配列のメモリを解放
    …………
}
```

6 イベントの検出と送信

Appクラスが「どのようにしてイベントを検出し，それを適切なウィジェットに送信しているか？」は，runメンバー関数を見ればわかる．

```
void App::run()
{
    int e;

    show();
    while ((e = getevent()) != CTRL_Q) {
```

```
            ............
            widgets[active]->handle_event(e);
        }
    }
```

runでは，まずshowを実行して全ウィジェットを表示した後，メインループに入っている．このループの中でイベントを検出し，アクティブなウィジェットにイベントを送信しているのである．このメインループを大雑把に書くと，次のようになる．

```
    while ((e = getevent()) != CTRL_Q) { ────────イベントの検出
        ............
        widgets[active]->handle_event(e);────────イベントの送信
    }
```

geteventがイベントを検出する関数である．第33章で説明したものと同じであるが，ここではAppのメンバー関数になっている．

```
    int App::getevent()
    {
        int c, c2;

        if ((c = getchar()) > 0x7f) {
            c2 = getchar();
            return (c << 8) | c2;
        }
        else
            return c;
    }
```

geteventの関数値は，漢字対応のキーコードである．その上位バイトが0ならASCII文字，そうでなければ漢字である．

そのgeteventの関数値を変数eに受け取る．それがCTRL_Qならループを抜けてプログラムを終了する（そのときデストラクタが実行される）．そうでなければ，activeが指すウィジェットのhandle_eventが呼び出される．これにより，イベントeが，そのウィジェットによって処理される．

7 ホットキー

runのメインループでは，イベントがウィジェットに渡される前に，前処理が行われている．これは，ホットキーを処理するためである．つまり，ホットキーは，イベントがウィジェットに渡される前にフックすることによって実現されている．

まず，チェックされるのがCTRL_Qである．これにより，アプリケーションを終了するかどうかが判定される．

```
while ((e = getevent()) != CTRL_Q) { ────── CTRL_Qのチェック
```

次に，メンバー関数のhotkeyが処理される．

```
while ((e = getevent()) != CTRL_Q) {
    if (hotkey(e))
        continue;
    ............

}
```

hotkeyはAppの仮想関数である．Appでは何もしないでreturnしている．アプリケーションでホットキーを定義する場合は，この関数をオーバーロードする（第3.4節参照）．

```
class App {
    ............
public:
    ............
    virtual int hotkey(int e) { return 0; };
    ............
};
```

hotkeyがそのイベントを処理した場合は，もうそのイベントをウィジェットに渡す必要はない．hotkeyがそのイベントを処理したか，または無視したかは関数値を見ればわかる．関数値が1（真）の場合は，有効なホットキーだったので，continueで次のイベントに移る．

```
if (hotkey(e))
    continue; ────────── 次のイベントへ
// まだイベントは処理されていない
```

もう1つ，デフォルトのホットキーが定義されている．それは，CTRL_Nである．このキーが押されると，次のウィジェットにフォーカスが移動する．

```
while ((e = getevent()) != CTRL_Q) {
    if (hotkey(e))
        continue;
    if (e == CTRL_N)  ──────── CTRL_Nのチェック
        nextwidget();
    else if (next > 0)
        widgets[active]->handle_event(e);
}
```

nextwidgetのコードは，次のようになっている．

```
// 次のウィジェットにフォーカスを当てる
void App::nextwidget()
{
    if (++active == next)
        active = 0;      // 最後の項目なら先頭の項目に戻る
    if (next > 0)
        widgets[active]->setfocus();
}
```

8 全リスト

アプリケーションフレームワークの全リストを示す．

8.1◆app.h

```
// app.h -- Application Frame work
//            by K. Tsukagoshi
//            1998.7.16 - 10.29

#ifndef APP
```

```
#define APP

const int CTRL_B = 0x02;
const int CTRL_D = 0x04;
const int CTRL_F = 0x06;
const int CTRL_H = 0x08;
const int CTRL_N = 0x0e;
const int CTRL_Q = 0x11;
const int CTRL_U = 0x15;
const int CTRL_W = 0x17;

const int INDENT = 10;

// ウィジェットのための仮想基本クラス
class Widget {
protected:
    int home_x;         // ホーム座標
    int home_y;
public:
    Widget(int x, int y);
    virtual ~Widget() {};
    virtual void show() = 0;
    virtual void setfocus() = 0;
    virtual void handle_event(int e) = 0;
};

// エディットボックスクラス
// 編集バッファ
//      先頭アドレス          : buf
//      入力可能文字数        : width
//      バッファのサイズ      : width + 1 ('\0'が必要)
//      バッファ上の入力位置  : col
//      画面上の座標          : home_x + col
class EditBox : public Widget {
  char label[20];           // 編集領域の左に表示するラベル
    int width;              // 最大入力可能文字数
    int label_x;            // ラベルの先頭
                            // 編集領域の先頭はhome_x
    char *buf;              // 編集バッファ
    int col;                // buf上の次の入力位置

    void cls();
    void empty();
    void inschar(int e);
```

```
    void bschar();
    void delchar();
    void moveleft();
    void moveright();
public:
    EditBox(int x, int y, char *l, int w);
    ~EditBox();
    void show();
    void setfocus();
    char *get_text();
    void handle_event(int e);
};

// アプリケーションのカプセル化
class App {
private:
    void init_curses();
    void show();
    void nextwidget();
    int getevent();

protected:
    int nwidgets;           // 管理するウィジェットの数
    Widget **widgets;       // ウィジェットの配列
    int active;             // アクティブなウィジェットのインデックス
    int next;               // 次にaddするウィジェットのインデックス
public:
    App(char *h, char *f, int n);
    ~App();
    void add_widget(Widget *pw);
    virtual int hotkey(int e) { return 0; };
    virtual void run();
};

#endif
```

8.2◆app.C

```
// app.C -- Application Frame work
//              by K. Tsukagoshi
//              1998.7.16 - 10.29
```

```
#include <curses.h>
#include <signal.h>                     // for signal
#include <string.h>                     // for strcpy
#include "app.h"

// ------------
// Widget class
// ------------

// x, y :ホーム座標
Widget::Widget(int x, int y)
{
    home_x = x, home_y = y;
}

// -------------
// EditBox class
// -------------

// ラベルと編集領域の初期化
EditBox::EditBox(int x, int y, char *l, int w)
    : Widget(x + strlen(l) + 2, y)
{
    strcpy(label, l);                   // ラベルの初期化
    label_x = x;

    width = w;                          // 編集領域の初期化
    buf = new char[width + 1];
    *buf = '\0';
    col = 0;
}

EditBox::~EditBox()
{
    delete [] buf;
}

// 編集領域をスペースでクリアする
// バッファには手をつけない
void EditBox::cls()
{
    attrset(WA_UNDERLINE);
    move(home_y, home_x);
    for (int i = 0; i < width; i++)
```

```
        addch(' ');
    refresh();
}

// 編集領域を表示領域,バッファともに空にする
void EditBox::empty()
{
    cls();
    *buf = '\0';
    col = 0;
    move(home_y, home_x);
}

// エディットボックスを表示する
void EditBox::show()
{
    attrset(WA_NORMAL);
    move(home_y, label_x);
    addstr(label);
    attrset(WA_UNDERLINE);
    cls();
}

// フォーカスを得たときの処理
void EditBox::setfocus()
{
    move(home_y, home_x);
    col = 0;
    attrset(WA_UNDERLINE);
    refresh();
}

// 現在のテキストを返す
char *EditBox::get_text()
{
    return buf;
}

// 1文字挿入
void EditBox::inschar(int e)
{
    char *p;
    int e1, e2;
```

```
    if (e > 0x7f) {  // 漢字の場合
        if (strlen(buf) >= width - 1) {
            beep();
            return;
        }
        else {
            // 後ろからコピーして空き領域を作る
            p = buf + strlen(buf);  // pは\0を指す
            while (p >= (buf + col)) {
                *(p + 2) = *p;
                p--;
            }
            e1 = (e >> 8);
            e2 = (e & 0xff);
            buf[col++] = e1;
            buf[col++] = e2;
        }
    }
    else {              // ASCIIの場合
        if (strlen(buf) >= width) {
            beep();
            return;
        }
        else {
            // 後ろからコピーして空き領域を作る
            p = buf + strlen(buf);  // pは\0を指す
            while (p >= (buf + col)) {
                *(p + 1) = *p;
                p--;
            }
            buf[col++] = e;
        }
    }
    move(home_y, home_x);
    addstr(buf);
    move(home_y, home_x + col);
}

// カーソル上の1文字を削除
void EditBox::delchar()
{
    char *p;

    if (col >= strlen(buf)) {    // バッファの右端にいるときは削除できない
```

```
        beep();
        return;
    }
    if (buf[col] & 0x80) {  // 漢字
        p = buf + col + 2;
        while (*p) {
            *(p - 2) = *p;
            p++;
        }
        *(p - 2) = '\0';
    }
    else { // ASCII
        p = buf + col + 1;
        while (*p) {
            *(p - 1) = *p;
            p++;
        }
        *(p - 1) = '\0';
    }
    cls(); // ひらがなを正しく消すには一度refreshする
    move(home_y, home_x);
    addstr(buf);
    move(home_y, home_x + col);
}

// 左の1文字を削除
void EditBox::bschar()
{
    if (col <= 0)          // バッファの左端にいるときは削除できない
        beep();
    else {
        moveleft();
        delchar();
    }
}

// 左に移動
void EditBox::moveleft()
{
    if (col <= 0)          // バッファの左端にいるときは移動できない
        beep();
    else if (buf[--col] & 0x80) {  // 漢字
        --col;
        move(home_y, home_x + col);
```

```
    }
    else  // ASCII
        move(home_y, home_x + col);
}

// 右に移動
void EditBox::moveright()
{
    if (!buf[col])                  // 最後の文字の右にいるときは移動できない
        beep();
     else if (buf[col++] & 0x80) {  // 漢字
        col++;
        move(home_y, home_x + col);
    }
    else  // ASCII
        move(home_y, home_x + col);
}

// エディットボックスのイベントハンドラ
void EditBox::handle_event(int e)
{
    if (e == CTRL_B)
        moveleft();
    else if (e == CTRL_F)
        moveright();
    else if (e == CTRL_H)           // 左の文字を削除
        bschar();
    else if (e == CTRL_D)           // カーソル上の文字を削除
        delchar();
    else if (e == CTRL_U)           // 編集領域を空にする
        empty();
    else if (e < 0x20)              // その他のコントロールコードは無効
        beep();
    else                            // 1文字挿入追加
        inschar(e);
    refresh();
}

// ---------
// App class
// ---------

// h :ヘッダ, f :フッタ, n :使用するウィジェットの数
App::App(char *h, char *f, int n)
```

```
{
    init_curses();

    move(1, INDENT);                //  ヘッダの表示
    attrset(WA_REVERSE);
    addstr(h);
    move(LINES - 2, INDENT);        //  フッタの表示
    attrset(WA_UNDERLINE);
    addstr(f);

    attrset(WA_NORMAL);
    refresh();

    nwidgets = n;
    widgets = new (Widget *)[nwidgets];
    active = next = 0;
}

App::~App()
{
    for (int i = 0; i < next; i++)
        delete widgets[i];
    delete [] widgets;

    clear();                        //  refresh()が必要
    refresh();
    endwin();                       //  cursesの使用を終了
}

// cursesの関係の初期化
void App::init_curses()
{
    initscr();          //  cursesの使用開始
    start_color();      //  色の使用を可能にする
    raw();              //  rawモードにする
                        //  rawモードではnoecho(), nonl()は不要
}

//  指定されたウィジェットをアプリケーションに追加する
void App::add_widget(Widget *pw)
{
    widgets[next++] = pw;
}
```

```
// 全ウィジェットを表示し，
// アクティブなウィジェットにフォーカスを当てる
void App::show()
{
    for (int i = 0; i < next; i++)
        widgets[i]->show();
    if (next > 0)
        widgets[active]->setfocus();
}

// 次のウィジェットにフォーカスを当てる
void App::nextwidget()
{
    if (++active == next)
        active = 0; // 最後の項目なら先頭の項目に戻る
    if (next > 0)
        widgets[active]->setfocus();
}

// キー入力を行う
// 漢字の場合は2バイトにして返す
// 上位バイトが0ならASCIIコード
// 半角2バイト文字には非対応
int App::getevent()
{
    int c, c2;

    if ((c = getchar()) > 0x7f) {
        c2 = getchar();
        return (c << 8) | c2;
    }
    else
        return c;
}

// アプリケーションの実行
void App::run()
{
    int e;

    show();
    while ((e = getevent()) != CTRL_Q) {
        if (hotkey(e))
            continue;
```

```
        if (e == CTRL_N)
            nextwidget();
        else if (next > 0)
            widgets[active]->handle_event(e);
    }
}
```

Index

Index

あ行

か行

Index

カバーデザイン❖ 小倉一夫
本文レイアウト❖ 堀田繁範（Holly）
編集❖ 跡部和之

ソフトウェアテクノロジー 25
［決定版］はじめての C++（シープラスプラス）

平成11年10月10日　初版　第1刷発行
平成12年12月 1 日　初版　第6刷発行

著　者　塚越一雄（つかごしかずお）
発行者　片岡　巖
発行所　株式会社技術評論社
東京都新宿区愛住町8番地8
電話　03-3225-2300　営業部
03-3225-3293　編集部
印刷／製本　大日本印刷株式会社

定価はカバーに表示してあります

ISBN4-7741-0843-X　C3055

Printed in Japan

ISBN4-7741-0843-X　C3055　¥2680E
定価（本体2680円＋税）
M-cord 208251
9784774108438
1923055026803

Software Technology
25

決定版
はじめてのC++

塚越一雄／著

技術評論社